Ewon® BY HMS NETWORKS

Programming Reference Guide

REFERENCE GUIDE

RG-0006-01 1.9 en-US ENGLISH

HMS Connecting Devices™

Important User Information

Disclaimer

The information in this document is for informational purposes only. Please inform HMS Networks of any inaccuracies or omissions found in this document. HMS Networks disclaims any responsibility or liability for any errors that may appear in this document.

HMS Networks reserves the right to modify its products in line with its policy of continuous product development. The information in this document shall therefore not be construed as a commitment on the part of HMS Networks and is subject to change without notice. HMS Networks makes no commitment to update or keep current the information in this document.

The data, examples and illustrations found in this document are included for illustrative purposes and are only intended to help improve understanding of the functionality and handling of the product. In view of the wide range of possible applications of the product, and because of the many variables and requirements associated with any particular implementation, HMS Networks cannot assume responsibility or liability for actual use based on the data, examples or illustrations included in this document nor for any damages incurred during installation of the product. Those responsible for the use of the product must acquire sufficient knowledge in order to ensure that the product is used correctly in their specific application and that the application meets all performance and safety requirements including any applicable laws, regulations, codes and standards. Further, HMS Networks will under no circumstances assume liability or responsibility for any problems that may arise as a result from the use of undocumented features or functional side effects found outside the documented scope of the product. The effects caused by any direct or indirect use of such aspects of the product are undefined and may include e.g. compatibility issues and stability issues.

Table of Contents

Page

This page intentionally left blank

1 Preface

1.1 About This Document

This document describes the BASIC scripting and all its possibilities when running on an eWON Flexy.

This document is an evolution of the former RG-002: Programming Reference Guide. The content written in this document can be applied only on eWON Flexy running a firmware version higher or equal to v12.2s0.

Other eWON devices compatible with BASIC (eWON CD and eWON Flexy) must refer to:

- Firmware of the device <= v8.1s4 : RG-002: Programming Reference Guide
- v8.1s4 > firmware of the device < v12.2s0 : RG-0006–00: Programming Reference Guide

For additional related documentation and file downloads, please visit developer.ewon.biz.

1.2 Document history

Version	Date	Description
1.0	2016-05-24	ADDED: New version of RG-0002
1.1	2016-20-12	CHANGED: Template ADDED: MQTT Command ADDED: LOGGROUPIO
1.2	2018-05-03	CHANGED: Typo in the ONDATE function CHANGED: COMCFG parameters CHANGED: Overall typo, examples corrections, ...
1.3	2018-09-12	Added: *FS, p. 35*
1.4	2019-01-23	Changed: *ONWAN, p. 79*
1.5	2020-09-07	Updated: *RESPONSEHTTPX, p. 55*
1.6	2020-09-16	Updated: *LOGGROUPIO, p. 62* Updated: *LOGIO, p. 62*
1.7	2020-11-30	Changed: Command of *S2* in *REQUESTHTTPX, p. 54*
1.8	2020-02-09	Changed: *LOGGROUPIO, p. 62* Changed: *HTTPX, p. 54* Changed: *TagName Variable, p. 20*
1.9	2021-07-08	Added: *Cyclic Section as of Firmware v14.5, p. 5* Changed:*LOGGROUPIO, p. 62* and *LOGIO, p. 62*

1.3 Related Documents

Document	Author	Document ID
comcfg.txt	HMS	KB-0050–00
config.txt	HMS	KB-0052–00

1.4 Trademark Information

eWON® is a registered trademark of HMS Industrial Networks SA. All other trademarks mentioned in this document are the property of their respective holders.

1.5 Cyclic Section as of Firmware v14.5

As of firmware version 14.5, the cyclic section — of the BASIC IDE — is no longer displayed on the Ewon GUI interface.

Many clients used the cyclic section in a wrong way which, depending on the script used, could have a negative impact on the overall Ewon behavior.

For most of the typical basic script use cases, you shouldn't consider the cyclic section.

If for specific reason you need to add the cyclic section, then you'll have to perform as follows:

1. Download the program.bas file from the Ewon device (using FTP or the Files Transfer menu).
2. Add the following lines inside the downloaded file:

```
Rem --- eWON start section: Cyclic Section
eWON_cyclic_section:
Rem --- eWON user (start)
// There must be at least one instruction (or comment) to keep
// the cyclic section displayed inside the Basic IDE
Rem --- eWON user (end)
End
Rem --- eWON end section: Cyclic Section
```

3. Re-upload the program.bas file on the Ewon device using FTP.

2 BASIC language definition

2.1 Introduction

The program of the eWON is based on syntax close to the BASIC, with many specific extensions.

BASIC scripting is possible thanks to the BASIC IDE available on the web interface of the device.

In this document, the following convention (if not indicated otherwise) is used to represent the parameters:

Parameter	Type
E1, E2	Integer
S1, S2	String
F1, F2	Real
CA	Character (if string passed, first char is evaluated)

2.2 Program Flow

It is important to understand how the device executes its program!

There's a difference between the record and the execution of the program within the device: the unit has a program task that extracts BASIC requests from a queue and executes those requests.

A request can be:

- A single command: *myVar=1*
- A branch to a label: *goto myLabel*
- A list of commands such as a program block

In the first case, the command is executed then the BASIC task is ready to execute the next request.

In the second case, the BASIC task goes to label *myLabel* and executes what's inside until the *END* command appears or until an error occurs.

Suppose the device hasn't any program but an *Init Section*, a *Cyclic Section* and a custom section labeled *myNew Section* are created:

Init Section

```
CLS
myVar = 0
```

Cyclic Section

```
FOR V% = 0 TO 10
  myVar = myVar + 1
NEXT V%
PRINT myVar
```

myNew Section

```
myNew Section:
myVar = 0
PRINT "myVar is reset"
```

If the corresponding program.bas file is downloaded using an FTP client, the following code will be obtained:

```
Rem --- eWON start section: Cyclic Section
eWON_cyclic_section:
Rem --- eWON user (start)
FOR V% = 0 TO 10
  myVar = myVar + 1
NEXT V%
PRINT myVar
Rem --- eWON user (end)
End
Rem --- eWON end section: Cyclic Section
Rem --- eWON start section: Init Section
eWON_init_section:
Rem --- eWON user (start)
CLS
myVar = 0
Rem --- eWON user (end)
End
Rem --- eWON end section: Init Section
Rem --- eWON start section: myNew Section
Rem --- eWON user (start)
mynew_section:
myVar = 0
PRINT "myVar is reset"
Rem --- eWON user (end)
End
Rem --- eWON end section: myNew Section
```

The code written in the BASIC IDE is displayed, but the device also added some remarks and labels to allow the modification and provide program flow control.

The IDE has added an *End* statement at the end of each section to prevent the program from continuing to the next section. The example also shows that each label is global to the whole program and should not be duplicated.

There is not correlation between the name of the section and the label used in that section.

The section name is only a method to organize program listing during modification. It can contain spaces while the program labels can not.

When the program starts (i.e: by clicking the *RUN* button from the top menu of the IDE), the device posts 2 commands in the queue:

BASIC Queue – 1

Queue position	Content	Type
...		
3		
2	goto ewon_cyclic_section	CYCLIC_SECTION
1	goto ewon_init_section	INIT_SECTION

The BASIC task of the device will read the request, from the queue, that has the lowest index and will execute it until an *End* is found or until an error occurs.

The first command is *GOTO ewon_init_section*. The following lines will be executed:

```
Rem --- eWON start section: Init Section
eWON_init_section:
Rem --- eWON user (start)
CLS
myVar = 0
Rem --- eWON user (end)
End
Rem --- eWON end section: Init Section
```

The *End* command on the line before last will end the program and the BASIC task will check in the queue for a new request:

BASIC Queue – 2		
Queue position	**Content**	**Type**
...		
3		
2		
1	goto ewon_cyclic_section	CYCLIC_SECTION

The first available command is *goto ewon_cyclic_section*, it will also be executed until the *End* command is found. When this *End* is executed the BASIC task will detect that the section that just run was a *CYCLIC_SECTION* and will then post a new *goto ewon_cyclic_section* request in the queue.

This explains how the program is continuously executed (and forever) as long as the BASIC is in *RUN* mode.

There are a number of actions that can be programmed to occur upon event, like ONTIMER:

```
TSET 1,10
ONTIMER 1, "goto myLabel"
```

If the above lines were in the *Init Section*, it would start a timer #1 with an interval of 10 seconds and program a *goto myLabel* request when timer #1 elapses.

When the *ONTIMER* occurs, the device posts the *goto myLabel* request in the BASIC queue.

BASIC Queue – 3		
Queue position	**Content**	**Type**
...		
3		
2	goto myLabel	
1	goto ewon_cyclic_section	CYCLIC_SECTION

When the *CYCLIC SECTION* will be finished, the timer request will be extracted from the queue and then executed. If the *CYCLIC SECTION* takes a long time to execute, then the timer can elapse more than once during the execution of the *CYCLIC SECTION* resulting in more timer action to be posted in the queue:

BASIC Queue – 4		
Queue pos ition	**Content**	**Type**
...		
5		
4	goto myLabel	
3	goto myLabel	
2	goto myLabel	
1	goto ewon_cyclic_section	CYCLIC_SECTION

The BASIC queue can hold more than 100 requests but if *TIMER* goes too fast or if other events such as *ONCHANGE* are used then the queue can overflow. In that case, an error is logged in the events file and requests are dropped.

The *ONTIMER* request is not executed with the exact precision of a timer, depending on the current load of the BASIC when the timer elapses.

When an ASP block needs to be executed because the device must deliver a web page to a client, the ASP block is also put in the queue.

As an example, if an ASP block contains the following lines:

```
FromWebVar = Var1!
PRINT #0;TIME$
```

Then the queue will reflect the following:

BASIC Queue – 5		
Queue pos	**Content**	**Type**
...		
3	FromWebVar = Var1! PRINT #0;TIME$	
2	goto MyLabel	
1	goto ewon_cyclic_section	CYCLIC_SECTION

If a request in the queue contains more than 1 BASIC line, the block is appended to the end of the program as a temporary section:

```
ewon_one_shot_section:
fromWebVar = Var1
PRINT #0;TIME$
END
```

The temporary label is called *goto ewon_one_shot_section*. When the execution is done, the temporary section is deleted from the program.

As a consequence, the following applies:

- Any global variable or label can be used in *remote.bas* file or ASP blocks; subroutines can be called in the ASP blocks and can share common variables with the program.
- If a section is being executed when the ASP section is posted, all the requests in the queue must first be executed. This may have an impact on the responsiveness of the website when ASP is used.
- When using ASP; it is recommended to group the blocks in order to avoid posting too many different requests in the queue. By doing so, queue extraction and BASIC context switches will be reduced.
- If a big amount of or long ASP requests are posted to the BASIC via the web server, it may slow down normal execution of the BASIC.

- Sections are never interrupted by other sections: this is always true! When a program sequence is written, it will never be broken by another execution (of timer, web request or anything else).

2.2.1 Character String

A character string can contain any set of characters. When creating an alphanumeric string with a quoted string the ‘ ‘ or “ “ delimiter must be used.

A character string can be stored either in an alphanumeric type variable or in an alphanumeric variable array.

Example of 3 valid strings

```
"abcd"
’abdc’
"abc‘def’ ghi "
```

2.2.2 Command

A command is an instruction that has none or several comma (,) separated parameters.

```
GOTO Label
PRINT
CLS
SETSYS TAG, "name","Power"
SETSYS TAG, "SAVE"
```

> **!** There are 2 exceptions to the comma separator: *PRINT* and *PUT*.

2.2.3 Integer

An integer is a number between -2147483648 and +2147483647 which be stored in an integer variable.

When a parameter of integer type is specified for a function or a command and this variable is actually of real type, the device converts automatically the real value to an integer value.

When the expected value is of integer type and the transmitted value is a character string, the device generates an error.

2.2.4 Real

A real number is a number in floating point representation of which value is between -3.4028236 10E38 and +3.4028234 10E38. Value of this type can be stored in a variable of real type or in an array of reals.

A real number has approximately 7 significant digits. This means that conversion of a number with more than 7 significant digits to real will lead to a lost in accuracy.

When a function expects a real number and an integer is transmitted, the device converts automatically the integer into a real value.

If the function awaits a real and a character string is passed, the eWON generates an error.

The device uses IEEE 754 single precision representation (32 bits). The fraction is coded on 23 bits, which represents about 7 significant digits. But on the web interface of the device, the

values of the tags are displayed only with 6 digits. If a tag is used in BASIC scripting, the 7 significant digits are then applied.

2.2.5 Alphanumeric Character

An alphanumeric character is one of the ASCII characters. Each ASCII character has a numerical representation between 0 and 255.

The *ASCII* function returns the ASCII code of a character, and the *CHR$* function converts the ASCII code to a string of a single character.

2.3 Function

A function is a BASIC command having none or several parameters and returning a result that can be of integer, real or string type.

```
ASCII "HOP"
GETSYS TAG, "NAME"
PI
```

2.3.1 Function Declaration

To declare a function, 2 keywords are needed:

- **FUNCTION**
  It is used to start the function definition and is followed on the same line by the function name which length must be greater than one character.
- **ENDFN**
  It is used to end the function definition.

Example 1: How to Declare a Function

```
FUNCTION my_function   // function definition begins
PRINT "my_string"
ENDFN
```

2.3.2 Function return value

The function return value is specified by using the following function name convention:

- If the function returns an integer: *Function my_function%*
- If the function returns a string: *Function my_function$*
- If the function returns a float: *Function my_function*

To specify the return value of a function, an implicit variable is created automatically based on the function name. When the function exits, the return value is the last value of this variable.

Example 2: Return Value of a Function

```
FUNCTION my_function
  $my_function = 1
  $my_function = $my_function + 1
  PRINT "my_string"
ENDFN
```

This example prints *my_string* in the console but the return value is 2.

2.3.3 Keyword “return” inside Functions

The keyword *return* can be used at any place inside a function to end it.

Example 3: Use of the Return Keyword

```
FUNCTION my_function
  IF (global_var%=1) THEN
    $my_function = 1.0
    RETURN
  ENDIF
  $my_function = 0.0
ENDFN
```

The current value of the *RETURN* ($FunctionName) will be returned just as if the *ENDFN* was reached.

2.3.4 Function Parameters

Parameters can be defined and applied to a function. These parameters need to be typed (same way as functions).

Properties of these parameters:

- Parameters are put between parenthesis and separated by a coma.
- Parameters are, by default, passed by value.
- Parameters type is deduced by the naming convention:
  - For string type: *$* at the end
  - For integer type: *%* at the end
  - For float type: nothing at the end
- When parameters are arguments passed by reference, they are labeled as:
  - *@$name$* for string type
  - *@$name* for integer and float type
  - *@$name%* is **not** supported
- Parameters are local variables in the function scope.
- These function parameters don't exist outside the function.

To clarify the distinction with standard variables: every parameter variable begins with *$* in the declaration and inside the function. This allows the manipulation of global and local variable with the same name without mistaking.

Example 4: Parameters of a Function

```
FUNCTION my_function($param1, $param2%, $param3$)
  $my_function = $param2% + $param1 + 1
```

```
ENDFN
PRINT @my_function(3, 3, "3")
```

2.3.5 Function Call

To call a function, the @ character precedes the function name and the parameters values are put between parenthesis. If there is no parameters, parenthesis may be omitted.

Example 5: How to Call a Function

```
FUNCTION my_function($param1)
  PRINT "call of [my_function] with param [";$param1;"]"
ENDFN

FUNCTION my_function2()
  PRINT "my_function2()"
ENDFN

FUNCTION my_function3
  PRINT "my_function3()"
ENDFN

@my_function(3)
@my_function2     // call of a function without parenthesis nor parameters
@my_function3()   // call of a function without parameters
```

Float and integer parameters must be handled with precaution. If a float is given as an integer parameter (or the other way around), an implicit cast will occur.

Example 6: Float / Integer error

```
FUNCTION my_function($param1%)
  PRINT "call of [my_function] with param [";$param1%;"]"
ENDFN

@my_function(3)    // This is OK
@my_function(3.4)  // This transformed the float into an integer
```

2.3.6 Passing Arguments by Reference

By default, the parameters are passed by value.

This means that side effects can't be executed. But sometimes, side effects are useful (i.e: a function that returns 3 values).

If the parameter is preceded by '@', they will be passed by reference. It can then be used as a normal parameter inside the function.

The only difference compared to a normal parameter (passed by value) is that changes made inside the function will be visible outside this function.

Example 7: Function with arguments by reference

```
FUNCTION my_function(@$param1,@$param2,@$param3$)
  $param1 = $param1 * 2
  $param2 = $param2 * 2
  $param3$ = "my_function_string"
ENDFN

v1 = 1.5
v2% = 2
v3$ = "my_string"

@my_function(v1, v2%, v3$)

PRINT v1    // Prints 3.00
PRINT v2%   // Prints 4
PRINT v3$   // Prints my_function_string
```

2.3.7 Recursive Function Call

A function can be called inside an already existing function.

Example 8: Function in a Function

```
FUNCTION exp($x, $n)
  IF ($n = 1) then
    $exp = $x
  ELSE
    IF ($n mod 2 = 0) THEN
      $exp = @exp($x * $x, $n / 2)
    ELSE
      $exp = $x * @exp($x * $x, ($n - 1) / 2)
    ENDIF
  ENDIF
ENDFN

PRINT @exp(3, 3)
```

2.4 Label

To use the *GOTO* and *GOSUB* commands, *labels* need to be defined.

A label is a name beginning a line and ended by a colon ":". The label name doesn't accept any space character.

The *GOTO* / *GOSUB* instruction uses the label name (without the colon) as parameter.

Example 9: Use of Label

```
GOTO "myLabel"
myLabel:
PRINT "Hello World"
```

2.4.1 Local Label

Sometimes, it's useful to have labels only inside a function to ease the flow control but without polluting the name spaces of the program.

To solve this, local labels can be defined in functions.

Example 10: Local Label

```
FUNCTION test_label():
  $a% = 1
  GOTO $exit
  $a% = 2
  $exit:
  $test_label = $a%
ENDFN
PRINT @test_label()    // Prints 1
```

A *GOTO* can be used inside the function to move to the *$exit:* label. Outside the function, this label doesn't exist.

2.5 Operators Priority

When these operations appear in expressions, they have the following priority:

1. Bracket terms
2. All functions except NOT and – (inversion)
3. Inversion of sign -
4. ^, *, /, MOD (modulo function)
5. +, -
6. =, >, <, <=, >=, <>
7. NOT, BNOT • AND, OR, XOR:

These expressions are ordered by decreasing order of priority.

*The operator ^ is the power operator such as 2^4 = 2*2*2*2*

2.6 Types of Variable

Variables typed as integer or as string can be defined with a long name. Long name variable are also applicable on array (i.e: *DIM arrayOfString(25,80)*)

Variable names are case insensitive (*myint%* and *MyInt%* are the same variable).

2.6.1 Integer Variable

```
abcdef%
```

abcdef%	The name of the variable, followed by the *%* sign which indicates a variable of integer type.

An integer variable can contain a number of integer type.

The variable name can contain alphabetical characters, numbers and "_" underscore, but name must begin with an alphabetical character.

Example 11: Syntax of Integer Variable

```
// unlimited number of variables
my_variable% = 1
// unlimited number of array of strings DIM A(25,80)
```

```
DIM arrayOfString(25,80)
// unlimited number of array of floats
DIM arrayOfFloat(25,80)
```

2.6.2 Real Variable

Syntax

```
abcdef
```

abcdef the name of the real variable

Variable names can contain up to 200 characters and are case insensitive: *AbCdEf* and *ABCDEF* represent the same variable.

The variable name can contain alphabetical characters, numbers and "_" underscore, but name must begin with an alphabetical character.

A real variable can contain a real number.

Example 12: Syntax of Real Variable

```
MyVar = 12.3  // valid
My_Var = 12.3  // valid
Var1 = 12.3  // valid
My Var = 12.3  // invalid
1Var = 12.3  // invalid
```

2.6.3 Alphanumeric String

Syntax

```
abcdef$
```

**abcdef$** the name of the variable,followed by the *$* sign which indicates a variable of string type

The name of the real variable can contain any number of characters. Its size is modified each time the content of the variable is modified.

It is possible to address parts of a string with the *TO* keyword:

Example 13: Use of the TO Keyword

```
A$(4 TO 10)    // returns a string from char. #4 to char. #10
A$(4 TO)            // returns a string from char. #4 until the end
A$(4 TO LEN(A$))     // same result as A$(4 TO)
```

2.6.4 Character Arrays

Syntax

```
DIM a$(E1 [, E2 [, E3 [,....]]][, I])
```

a$	the name of the array variable created.
E1 [, E2 [, E3 [,....]]]	the number of elements per dimension. *E2*, *E3*, *E4* are optional and are present if the array must have 2, 3, 4... dimensions.
l	the number of characters per element. If not supplied, the default value is 1

The number of dimensions is limited only by the memory size of the BASIC.

When the *DIM* command is called, the array is created and replaces any other *DIM* or variable existing with the same name. To erase an array:

- Use the *CLEAR* command which erases all variables
- Change the dimension of the array to a single element with another call to DIM in case the user doesn't want to clear everything but needs to release memory

An array named as *a$(E1,E2,E3)* and an alphanumeric variable named as *a$* can exist simultaneously.

A characters array contains E1*E2*E3 *... characters.

Example 14: Character Arrays

```
DIM A$(10,2,5)
A$(1,1) = "1-testing"
A$(1,2,1 to 4) = "2-testing"
A$(2,1,2 to) = "3-testing"
A$(3,1) = "4-testing"

PRINT A$(1,1)  // Outputs 1-tes
PRINT A$(1,2)  // Outputs 2-te
PRINT A$(2,1)  // Outputs 3-te
PRINT A$(3,1) // Outputs 4-tes
```

2.6.5 Real Arrays

The real arrays is also valid for integers as there is no dedicated integer arrays.

Syntax

```
DIM a(E1 [, E2 [, E3 [,....]]])
```

a	the name of the array variable created.
E1 [, E2 [, E3 [,....]]]	the number of real for the first dimension. *E2*, *E3*, *E4* are optional and are present if the array must have 2, 3, 4… dimensions.

The number of dimensions is limited only by the memory size of the BASIC.

When the *DIM* command is called, the array is created and replaces any other *DIM* or variable existing with the same name. To erase an array:

- Use the *CLEAR* command which erases all variables
- Change the dimension of the array to a single element with another call to DIM in case the user doesn't want to clear everything but needs to release memory

In order to assign a value, type *a(x, y, z) = value*.

An array named as *a(E1,E2,E3)* and a real variable named as *a* can exist simultaneously.

A real array contains E1*E2*E3 *... reals.

Example 15: Real Arrays

```
DIM d(5,5)
d(1,5) = 6.7
PRINT d(1,5)  // Outputs 6.70
```

2.6.6 Local Variables

Local variables are used to define variables visible only in the function scope.

The local variable needs to be preceding by the *$* character inside the function.

Example 16: Local Variables

```
FUNCTION a()
  $b = 3          // local variable b
  $a = $b + 3
ENDFN

exec:
print @a()        // here, @a() exists, but not $b.
```

2.7 TagName Variable

Syntax

```
TagName@
```

TagName	the name of a tag available in the device

Adding the “@” after the tag name allows direct access to the tag value. This syntax can be used for reading or writing to the tag.

Example 17: Reading a Tag Value

```
Tag1@ = 25.3
Tag2@ = Tag1@
IF (Tag3@ > 20.0) THEN
…
```

In some cases, you must use the `GETIO` or `SETIO` commands to build/retrieve the tag name in the program. Cases such as:

- Perform some repetitive operations.
- If a tag name begins with a number.
- If the tag name contains more than 42 characters.

Example 18: Using GETIO / SETIO

```
FOR i% = 1 to 10
  A$ = "Tag" + STR$(i%)
  SETIO A$, i%
NEXT i%
```

2.8 Tag Access

All the BASIC functions accessing tags can reference the tag by its name (without the @ like in the *TagName Variable, p. 20*), by its index or by its ID.

Tag Access: Name, Index or ID

Method	Parameter	Example	Example explanation
Tag name access	Tagname String	SETIO "TAG1",23.5	Set the value 23.5 in the Tag named TAG1
Index access	Negative Integer (or 0)	SETIO -2,23.5	Set the value 23.5 in the Tag at the INDEX 2 (the third entry in the var_lst.txt)
ID access	Positive Integer (>0)	SETIO 2,23.5	Set the value 23.5 in the Tag with the ID=2

If there are 6 tags defined in the config.txt file, each tag can be accessed by its index (-0 to -5) or by its ID (the first item of a tag definition) or finally by its name.

The ID of a tag is never used again by the device until this device is formatted (reset level 2).

2.9 Limitation of the BASIC

The BASIC script is limited by the memory allocated to it (128 k). Users have to share this memory space between the code and the data.

3 List of Keywords

The commands and functions used to program the device are listed below in alphabetical order.

The following commands or functions are available for any firmware version (with a minimum of v8.1s4) except specifically notified otherwise.

3.1 # (bit extraction operator)

Syntax

```
E1 # E2
```

E1	integer variable
E2	bit position (0 to 31)

Description

The # function is used to extract a bit from an integer variable (and from an integer only).

Example 19: Bit Extraction

```
i% = 5    // Binary 0101
a% = i%#0      // a%=1
b% = i%#1      // b%=0
c% = i%#2      // c%=1
```

3.2 // (comment)

Syntax

```
// Free text
```

Description

This command enables the insertion of a line of comment in the program. The interpreter does not consider the line.

The comment can be written on a new line or on a line already containing an instruction (command, function...).

Example 20: Insert a Comment

```
PRINT a%
// This line will not be taken into consideration
a% = 2    // Write a comment on the same line
```

Check also

REM, p. 92

3.3 ABS

Syntax

```
ABS E1
```

E1	can be a value or a tag name

Description

The function returns the absolute value of *E1*. If the value is negative, parenthesis *()* must be used.

Example 21: Absolute Value

```
ABS (-10.4)  // Returns 10.4
```

3.4 ALMACK

Syntax

```
ALMACK S1, S2
```

S1	the tag reference (tag name, ID or index)
S2	the name of the user that will acknowledge the alarm. If this field is filled with empty quotes " ", then the *adm* login is assumed for acknowledgment.

Description

The *ALMACK* function acknowledges the alarm status of a given tag. *ALMACK* returns the error "Operation failed (28)" if the tag is not in alarm.

Example 22: Acknowledge an Alarm

```
ALMACK "MyTag", "John"
```

3.5 ALSTAT

Syntax

```
ALSTAT S1
```

S1	the tag reference (tag name, ID or index)

Description

This function returns the *S1* tag alarm status. The possible returned values are:

ALSTAT: Possible Value for S1

Parameter	Type
0	No alarm
1	Pre-trigger: no active alarm but physical signal active
2	In alarm

ALSTAT: Possible Value for S1 (continued)

Parameter	Type
3	In alarm but acknowledged
4	Returns to normal but not acknowledged

Example 23: Get the Alarm Status of a Tag

```
a% = ALSTAT "MyTag"
```

3.6 AND

Syntax

```
E1 AND E2
```

Description

Do a bit-wise AND between *E1* and *E2*. Also have a look at the priority of the operators.

Example 24: Perform an AND operation

```
1 AND 2  // Returns 0
2 AND 2  // Returns 2
3 AND 1  // Returns 1
MyFirstTag@ AND 3  // Keeps first 2 bits
```

Check also

Operators Priority, p. 17; *OR, p. 85*; *XOR, p. 105*

3.7 ASCII26

Syntax

```
ASCII CA
```

Description

The function returns the ASCII code of the first character of the CA chain. If the chain is empty, the function returns 0.

Example 25: Get the ASCII code

```
a% = ASCII "HOP"  // Returns the ASCII code of the character H
```

Check also

CHR$, p. 25

3.8 BIN$

Syntax

```
BIN$ E1
```

Description

The function returns a string of 32 characters that represents the binary value of *E1*. It does not work on negative values.

Example 26: Get the Binary Value of an Integer as a String

```
a$ = BIN$ 5  // a$ equals "00000000000000000000000000000101"
```

Check also

HEX$, p. 53

3.9 BNOT

Syntax

```
BNOT E1
```

Description

This function returns the bitwise negation or one's complement of the integer *E1*.

Example 27: Get the Bitwise Negation of an Integer as a String

```
a% = 5
b% = BNOT a%
PRINT BIN$(b%)  // Prints "11111111111111111111111111111010"
```

Check also

Operators Priority, p. 17

3.10 CFGSAVE

Syntax

```
CFGSAVE
```

Description

This command writes the configuration of the device to flash. It is necessary after *SETSYS* command on *SYS*, *TAG* or *USER* records because using *SETSYS* modifies the configuration in memory.

The modification is effective as soon as the *SETSYS XXX, "save"* (where “XXX” stands for “SYS”, “USER” or “TAG”), but the config is not saved to the device flash file system.

Check also

GETSYS, SETSYS, p. 44

3.11 CHR$

Syntax

```
CHR$ E1
```

Description

The function returns a character string with only one character corresponding to the ASCII code of *E1*. *E1* must be contained in the 0..255 range.

Example 28: Get the Bitwise Negation of an Integer as a String

```
a$ = CHR$ 48          // a$ equals 0
b$ = CHR$(GETIO(MyTag))    // If MyTag = 32, then b$ will hold one space
```

Check also

ASCII26, p. 24

3.12 CLEAR

Syntax

```
CLEAR
```

Description

Erases all variables from the device. All DIM are erased. This command cannot be canceled.

3.13 CLOSE

Syntax

```
CLOSE I1
```

I1	the file number (# from 1 to 8)

Description

This command closes the file which number is*I1*. If the file is opened for write, it is actually written to the flash file system. The function can be called even if the file is not opened.

Check also

EOF, p. 29; *GET, p. 39*; *OPEN, p. 80*; *PUT, p. 88*

3.14 CLS

Syntax

```
CLS
```

Description

This command erases the virtual screen of the device, visible in the BASIC IDE debug panel.

Check also

PRINT – AT, p. 86

3.15 DAY

Syntax

```
DAY E1|S1
```

E1	a date in integer format: number of seconds since 1970-01-01
S1	a date in string format: “18/09/2003 15:45:30”

Description

This function returns an integer corresponding to the value of the day of the month (1 — 31) that matches a defined time variable.

If the function is called with a float variable as value, it will result in an error “invalid parameter”.

Example 29: Get the Day of the Month out of a Date Variable

```
a$ = TIME$
a% = DAY a$

b% = GETSYS PRG, ”TIMESEC”
a% = DAY b%
```

Check also

DOW, p. 28; *DOY, p. 28*; *MONTH, p. 64*; *WOY, p. 104*

3.16 DEC

Syntax

```
DEC S1
```

S1	the string to convert from HEX to DEC

Description

This function returns an integer corresponding to the hexadecimal value of parameter. The string is not case sensitive (i.e: a023fc = A023FC).

The string can be of any length.

Example 30: Convert from HEX to DEC

```
a$= HEX$(1234)
i% = DEC(a$)           // Now, I% = 1234
```

Check also

HEX$, p. 53

3.17 DIM

Description

The DIM function allows the creation of variables of array type. Two types of array are available:

- the characters arrays
- the real arrays.

Check also

Types of Variable, p. 17

3.18 DMSYNC

Syntax

```
DMSYNC
```

Description

The command has no parameter and triggers a Data Management synchronisation. If the Data Management has been configured on the device, this command will send the historical data to the Data Management system.

3.19 DOW

Syntax

```
DOW E1|S1
```

E1	a date in integer format: number of seconds since 1970-01-01
S1	a date in string format: “18/09/2003 15:45:30“

Description

This function returns an integer corresponding to the value of the day of the week (0 — 6; Sunday = 0) that matches a defined time variable.

If the function is called with a float variable as value, this will result in an error “invalid parameter”.

Example 31: Get the Day of the Week out of a Date Variable

```
a$ = TIME$
a% = DOW a$

b% = GETSYS PRG, "TIMESEC"
a% = DOW b%
```

Check also

DAY, p. 26; *DOY, p. 28*; *MONTH, p. 64*; *WOY, p. 104*

3.20 DOY

Syntax

```
DOY E1|S1
```

E1	a date in integer format: number of seconds since 1970-01-01
S1	a date in string format: “18/09/2003 15:45:30“

Description

This function returns an integer corresponding to the value of the current day in the year (0 — 365) that matches a defined time variable.

If the function is called with a float variable as value, this will result in an error “invalid parameter”.

Example 32: Get the Day of the Year out of a Date Variable

```
a$ = TIME$
a% = DOY a$

b% = GETSYS PRG, "TIMESEC"
a% = DOY b%
```

Check also

DAY, p. 26; *DOW, p. 28*; *MONTH, p. 64*; *WOY, p. 104*

3.21 DYNDNS

Syntax

```
DYNDNS
```

Description

The command has no parameter and asks a NO-IP dynamic PPP IP address update to the Dynamic DNS server set on the “Publish IP Address” web page of the device.

It will be used to synchronize a Dynamic DNS server such as No-IP with the eWON PPP IP address.

3.22 END

Syntax

```
END
```

Description

This command indicates the end of the program. It can also be used to stop the execution of a section. If the program is in RUN mode, this command will suspend the execution until another section is ready to run (*ONCHANGE*, *CYCLIC*...).

Example 33: Ending the Program

```
my-label:
PRINT “START”    // Prints START
END
PRINT “HELLO”    // This line is not printed
```

Check also

HALT, p. 53

3.23 EOF

Syntax

```
EOF E1
```

E1	a number (1 — 8) corresponding to a /usr file or an Export Block Descriptor

Description

This function returns *1* when *end of file* is reached. *EOF* always returns *1* with files opened for write.

EOF works only with *OPEN "file:..."* or *OPEN "exp:..." FileStream*.

Example 34: Knowing EOF Has Been Reached

```
PRINT "open file"
OPEN "file:/usr/myfile.txt" FOR TEXT INPUT AS 1

ReadNext:
IF EOF 1 THEN GOTO ReadDone
A$ = GET 1
PRINT A$
GOTO ReadNext

ReadDone:
PRINT "close file"
CLOSE 1
```

Check also

CLOSE, p. 26; *GET, p. 39*; *OPEN, p. 80*; *PUT, p. 88*

3.24 ERASE

Syntax

```
ERASE Filename|Keyword
```

Filename	the path to the file that needs to be erased
Keyword	Specific keyword to erase root files

Description

This command erases the specified file in the "/usr" directory. This means it doesn't work for a different directory than the "/usr" directory. Omitting "/usr/" before the filename will result in a syntax error.

The file and directory names are case sensitive.

However, to erase some root files, some special keywords have been integrated:

Keyword for ERASE command	
Keyword	**Description**
#ircall	To erase the ircall.bin file, then all historical logged data
#events	To erase the events.txt file, the diagnostics file.
#hst_alm	To erase the hst_alm.txt file, the alarms historical file.
#usr	To erase (and format) completely the "/usr" directory/partition
#sys	To erase (and format) completely the "/sys" directory/partition

Example 35: Erase a File

```
ERASE "/usr/myfile.shtm"
ERASE "#events"
```

Check also

RENAME, p. 93

3.25 FCNV

Syntax

```
FCNV S1, EType[, ESize, SFormat]
```

S1	the string to be converted
EType	the parameter determining the type of conversion
ESize	the size of the string to convert (can be shorter than the entire S1)
SFormat	the format specifier for the conversion

Description

Converts a string to a number (float or integer). The return value can be an IEEE float, an integer, a CRC16 or a LRC. The type of conversion is determined by the *EType* parameter.

EType for FCNV command	
Etype value	**Conversion type**
1	convert string (MSB first) to float
2	convert string (LSB first) to float
5	compute the CRC16 on string and return an integer
6	compute the LRC on string and return an integer
10	convert string (MSB first) to integer
11	convert string (LSB first) to integer
20	convert string to a float using an *SFormat* specifier
30	convert string to an integer using an *SFormat* specifier
40	convert time as string into time as integer

Check also

SFMT, p. 97

3.25.1 Convert from an IEEE Float Representation

The IEEE float representation use four bytes (32 bits).

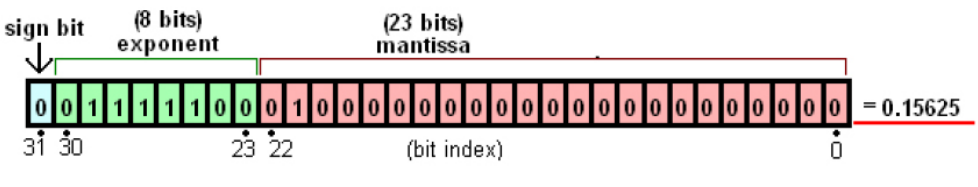

Fig. 1 Conversion to an IEEE Float

The string could be LSB (Least Significant Byte) first which will convert *a$(1 to 4)* to a float IEEE representation with MSB (Most Significant Byte) first.

```
FCNV a$, 1
a$(1)  // MSB which represents Exponent + Sign
...
a$(4)  // LSB which represents Mantissa
```

The string could also be MSB first which will convert *a$(1 to 4)* to a float IEEE representation with LSB first

```
FCNV a$, 2
a$(1)  // LSB which represents Mantissa
...
a$(4)  // MSB which represents Exponent + Sign
```

Example 36: Conversion to an IEEE Float Variable

```
ieee = 0.0
a$ = "1234"
a$(1) = Chr$(140)
a$(2) = Chr$(186)
a$(3) = Chr$(9)
a$(4) = Chr$(194)
ieee = FCNV a$,2
PRINT ieee // This will print -34.432176
```

3.25.2 Compute CRC16 of a String

Compute the Cyclic Redundancy Check (CRC) of the string.

CRC-16 uses the Polynomical 0x8005 (x16 + x15 + x2 + 1) with an init value of 0xFFFF.

Example 37: Computer CRC16 of a string

```
a$ = "My string"
c% = FCNV a$, 5
PRINT c%    // Prints 51608
```

3.25.3 Compute LRC of a String

Compute the LRC (Longitudinal Redundancy Check) of the string.

The LRC computation is the sum of all bytes modulo 256.

Example 38: Computer CRC16 of a string

```
a$ = "My string"
c% = FCNV a$, 6
PRINT c%    // Prints 125
```

3.25.4 Compute from an Integer Representation

Convert a string containing several bytes (1 to 4) in an integer value.

The integer representation could be LSB (Least Significant Byte) first or MSB (Most Significant Byte) first.

The *ESize* parameter is required as it indicates the size of the string to convert: 1, 2 ,3 or 4

```
// Convert a$(1 to 4) to an integer representation with MSB first
```

```
FCNV a$, 10, 4
a$(1)    // MSB
...
a$(4)    // LSB

// Convert a$(1 to 2) to an integer representation with MSB first
FCNV A$, 10, 2
a$(1)    // MSB
...
a$(4)    // LSB

// Convert a$(1 to 4) to an integer representation with LSB first
FCNV a$, 11, 4
a$(1)    // LSB
...
a$(4)    // MSB

// Convert a$(1 to 2) to an integer representation with LSB first
FCNV a$, 11, 2
a$(1)    // LSB
...
a$(4)    // MSB
```

Example 39: Convert from an Integer Variable

```
a$ = CHR$(1) + CHR$(4) + CHR$(2) + CHR$(0)
a% = FCNV a$, 10, 2
b% = FCNV a$, 11, 2
PRINT a%    // a% = 260
PRINT b%    // b% = 1025

c% = FCNV a$, 10, 3
d% = FCNV a$, 10, 4
PRINT c%    // c% = 66562
PRINT d%    // d% = 17039872
```

3.25.5 Convert String to a Float Using an SFormat Specifier

Convert a string with a float number (i.e: a$ = "153.24") to a float variable using a format specifier.

The *ESize* parameter is required as it is the size of the string to convert (*0* is to convert the whole string).

The *SFormat* parameter is required as it is the format specifier string and must be *"%f"*.

Example 40: Convert String to a Float

```
float_0 = FCNV "14.2115", 20, 0, "%f"  // float_0 = 14.2115
float_1 = FCNV "14.2115", 20, 4, "%f"  // float_1 = 14.2
float_2 = FCNV "-142.1e3", 20, 0, "%f"
```

3.25.6 Convert String to an Interger Using an SFormat Specifier

Convert a string with an integer number (i.e: a$ = "154" or a$ = "F0E1") to an integer variable using a format specifier.

The *ESize* parameter is required as it is the size of the string to convert (*0* is to convert the whole string).

The *SFormat* parameter is required as it is the format specifier string and can be *"%f"*:

- *"%d"* if the string holds a decimal number.
- *"%o"* if the string holds an octal number.
- *"%x"* if the string holds an hexadecimal bumber.

Example 41: Convert String to a Integer

```
a% = FCNV "1564", 30, 0, "%d"  // a% = 1564
a% = FCNV "1564", 30, 2, "%d"  // a% = 15
a% = FCNV "FE", 30, 0, "%x"  // a% = 254
a% = FCNV "11", 30, 0, "%o"  // a% = 9
```

3.25.7 Convert Time as String into Time as Integer

Convert a string holding a time in the format “dd/mm/yyyy hh:mm:ss” (ex: “28/02/2007 16:48:22”) into an integer holding the number of seconds since 1970-01-01 00:00:00.

Float value is not accurate enough to hold big numbers used to represent seconds since 1970-01-01, this leads to a lost of precision during time conversion.

Example 42: Convert Time String to Time Integer

```
a% = FCNV "24/04/2007 12:00:00", 40  // a% = 1177416000
a% = FCNV "01/01/1980 00:00:00", 40  // a% = 315532800
```

Check also

TIME$, p. 101

3.26 FOR - NEXT - STEP

Syntax

```
FOR a% = E1 TO E2 [STEP E3]
[Instructions]
NEXT a%
```

a%	an integer variable used as a counter. Variable must be a single character only (*ab%* is not allowed).
E1, E2 and E3	integer values / variables

Description

The instructions between the lines containing the FOR and the NEXT are executed until *a%* is outside the bounds of *(E1, E2)*. The loop is always executed at least once, even if *E1* is greater than *E2*.

During the first loop execution, *a%* equals *E1*.

FOR and *NEXT* commands cannot be on the same line of program.

Do not exit the *FOR/NEXT* loop by a *GOTO* statement because, in this case, after a certain number of executions, the memory of the device will be full.

If *STEP* is not mentioned, *a%* increases by 1

If *a%* is used inside a function, it should be used as a local variable.

Example 43: For Loop

```
FOR a% = 10 TO 20 STEP 2
  PRINT a%
NEXT a%
```

3.27 FS

The file system *FS*, allow the user to manage files in the *user* directory [for programs, data logging, etc.]

> **WARNING**
> It is recommend using this commands on an external SD card ext3 [eMMC, ext4] to avoid harms in the eWON device.

3.27.1 ls

Syntax

```
FS "ls", S1, E1
```

S1	The path to the directory that will be listed.
E1	The integer value representing the index.

Description

This function lists the content of a directory.

The “ls” command retrieves all the directories and files listed in the *S1* directory. It will then display the directory / file upon the *E1* index.

The *E1* index is sorted / based on the directory / file creation.

Example 44:

```
if /usr/sdext contains howdy, hello, hosts.txt
ls$ = FS "ls", "/usr/sdext", 0  // ls contains "pki"
ls$ = FS "ls", "/usr/sdext", 2  // ls contains "hosts.txt"
```

3.27.2 Touch

Syntax

```
FS "touch", S1
```

S1	The path to the file that will be checked.

Description

This function checks if a file exists or not. If the file doesn’t exist:

- it creates an empty file with the supplied name in *S1*
- it returns “1”

If the file does exist:

- it returns “-1”

This doesn’t apply to directories. This function doesn’t create (sub)directories in /usr, it can only check if they exist or not.

Example 45:

```
FS "touch", "/usr/sdext/f1.txt"
// returns -1 if "f1" exists
// returns 1 if "f1" doesn’t exist and creates it as a file
```

3.27.3 Size

Syntax

```
FS "size", S1
```

S1	The path to the file that will be checked.

Description

This function provides the size of a file in bytes. If the file doesn’t exist, the function returns “-1”.

This doesn’t apply to directories. This function doesn’t retrieve the size of a directory.

Example 46:

```
FS "size", "/usr/sdext/f1.txt"
// returns the file size of "f1"
// returns –1 if "f1" doesn’t exist.
```

3.27.4 Count

Syntax

```
FS "count", S1
```

S1	The path to the directory where items will be counted.

Description

This function gives you the number of items in a directory. It counts the files but also the directories.

Example 47:

```
FS "count", "/usr/sdext"
// returns the number of items contain in /usr/sdext
// returns 0 if S1 doesn’t countain any item
// returns –1 if S1 is invalid or doesn’t exists
```

3.27.5 isFile

Syntax

```
FS "isFile", S1
```

S1	The path to the file to perform the check.

Description

This function determines if *S1* is a regular file.

Example 48:

```
FS "isFile", "/usr/sdext/f1.txt"
// returns 1 if "f1" is a regular file
// returns -1 if "f1" is invalid (wrong filename, directory)
```

3.27.6 isDir

Syntax

```
FS "isDir", S1
```

S1	The path to the directory to perform the check.

Description

This function determines if *S1* is a directory.

Example 49:

```
FS "isDir", "/usr/sdext/d1"
// returns 1 if "d1" is a directory
// returns -1 if "d1" is not a directory
```

3.27.7 mkdir

Syntax

```
FS "mkdir", S1
```

S1	The path to the directory that must be created.

Description

This function creates a directory at *S1* and takes the directory path and returns the result.

To create a directory tree, the function needs to be called multiple times because it creates only a single level directory at a time.

Example 50:

```
FS "mkdir", "/usr/sdext/d1"
// returns -1 in case of error (multiple sub-directories)
// returns 1 in case directory was created
```

3.27.8 rm

Syntax

```
FS "rm", S1
```

S1	The path to the directory or file that should be deleted.

Description

This function deletes a directory or a file and takes a path and return the result.

It deletes one item at a time, so to delete a non empty directory, delete all the items in this directory first.

Example 51: It will return -1 in case of error, other wise it will return 1

```
FS "rm", "/usr/sdext/f1"
// returns 1 in case "f1" has been deleted
// returns –1 in case of error (director not empty, file doesn’t exist...)
```

3.27.9 cp

Syntax

```
FS "cp", S1, S2
```

S1	The path where the file is currently located.
S2	The path where the file should be copied.

Description

This function copies a file and its content from *S1* to a new destination *S2*. The source should be a valid regular file and the destination file name should not exist.

The directory to which the file will be copied must exist prior to the copy.

Example 52: It will return -1 in case of error (invalid filename, invalid destination, etc), other wise it will return 1

```
FS "cp", "/usr/sdext/f1", "/usr/sdext/f2"
// returns 1 if "f2" has been created and contains the content of "f1"
// returns –1 in case of error (non-existing directory, "f2" already exists, ...)
```

3.27.10 mv

Syntax

```
FS "mv", S1, S2
```

S1	The path to the file or directory that needs to be moved.
S2	The path to where the file or directory should be moved (embedded with the filename or file directory).

Description

This function moves a file or directory (and its sub-folder/files)*S1* to *S2* destination. The source must be a valid regular file and the destination must not exist.

The directory to which the file will be moved must exist prior to the transfer. Sub-directories are not created on the fly.

This function can also be used to rename a file inside the diferent/same directory.

Example 53:

```
FS "mv", "/usr/sdext/f1", "/usr/sdext/meme/f1"
// returns 1 in case of success
// returns -1 in case of error (invalid filename, destination, ...)
```

3.28 GET

The *GET* command works completely differently if the file is opened in binary or text mode.

The file syntax has been extended to allow access to the serial port and TCP | UDP socket.

The command description describes operation for

- /usr (text and binary modes)
- COM (always binary)
- TCP-UDP (always binary)

3.28.1 /usr in Binary Mode

Syntax

```
GET E1, E2|S1
```

E1	the file number (1 — 8)
E2	the number of bytes to read from the file
S1	the keyword on which the function will base its return value

Description

This function returns a string of character with the data read. It also moves the file read pointer to the character following the last character read (or to end of file).

- *Get 1, 1* returns maximum 1 character
- *Get 1, 5000* returns maximum 5000 characters
- *Get 1* without parameter is equivalent to *Get 1, 2048*

A keyword *S1* can be used instead of an integer *E2*. By using a keyword, the function returns file specific information.

***S1* Value**	**Return information**
SIZE	Total file size

Example 54: GET in /usr folder – Binary Mode

```
OPEN "file:/usr/myfile.bin" FOR BINARY INPUT AS 1
a$ = GET 1, 10  // Read 10 bytes
PRINT a$
CLOSE 1
```

3.28.2 /usr in Text Mode

Syntax

```
GET E1[, E2]
```

E1	the file number (1 — 4)
E2	the buffer size

Description

This function returns a string or a float according to the data read from the file. If this data is surrounded with quotes then it is returned as a string, otherwise it is returned as a float. The function will never return an integer

For string items, the single quote or double quotes can be used. The separator between items is the semicolon character.

When data is read from the file, it must be read in a buffer to be interpreted. The buffer must be able to hold at least the whole item and the CRLF at the end of the line if the item is the last of the line. The default buffer size is 1000 bytes, if the file contains items that might be bigger than 1000 bytes, *E2* parameter should be specified.

The function moves the file read pointer to the next item.

When a *CRLF (CHR$(13)+CHR$(10))* is found it is also skipped.

Example 55: GET in /usr folder – Text Mode

```
// myfile.txt content:
// 123;"ABC"
// 1.345;"HOP"

DIM a$(2,20)
DIM a(2)
OPEN "/myfile.txt"
FOR TEXT INPUT AS 1
I% = 1

ReadNext:
IF EOF 1 THEN GOTO ReadDone
a(I%) = GET 1
a$(I%) = GET 1
Ii% = i% + 1
GOTO ReadNext

ReadDone:
CLOSE 1
```

3.28.3 COM in Binary Mode

Syntax

```
GET E1, E2
```

E1	the file number
E2	the maximum number of bytes to read from the serial port

Description

This function returns a string with the data read from the serial port buffer. If there is no data to read from the buffer the returned string is empty.

If *E2* is specified and the buffer contains more than *E2* bytes, the function returns only with *E2* bytes.

If *E2* is specified and the buffer contains less than *E2* bytes, the function returns with the content of the buffer.

The function always returns immediately.

Attempting to use a serial port configured and occupied by an IO server is not allowed and returns an error.

Example 56: GET from COM – Binary Mode

```
OPEN "COM:2, ... AS 1"
a$ = GET 1, 100
CLOSE 1
```

3.28.4 TCP/UDP in Binary Mode

Syntax

```
GET E1, E2
```

E1	the file number returned by *OPEN*
E2	the maximum number of bytes to read from the socket

Description

This function returns a string with the data read from the TCP/UDP socket. If there is no data to read from the buffer, the returned string is empty.

If *E2* is specified and the buffer contains more than *E2* bytes, the function returns only with *E2* bytes.

If *E2* is specified and the buffer contains less than *E2* bytes, the function returns with the content of the buffer. If the other party has closed the socket or if the socket is in error at the TCP/IP stack level, the function exits with error

The function always returns immediately.

Check also

CLOSE, p. 26; *EOF, p. 29*; *OPEN, p. 80*; *ONERROR, p. 75*; *PUT, p. 88*

3.29 GETFTP

Syntax

```
GETFTP S1, S2[, S3]
```

S1	the name of the source file to retrieve from the FTP server
S2	the name of the destination file to write on the eWON
S3	the FTP server connection parameters. Formatted as *[user:password@]servername[:port][,option1]*

Description

This function retrieves a file on an FTP server and copies it on the device.

The source filename can include a path, built with slash “/” or backslash “\” depending of the FTP server. As the destination filename is on the device, its path must begin with a slash “/” and should include a path built with slash “/” as well.

The [option1] parameter from *S3* configures the mode of the communication. If [option1] is omitted, the device will connect in active mode. Possible values are:

- 1: passive mode
- 0: active mode

In the case *S3* is not provided, the FTP server parameters on the main configuration page from the web interface of the device will be used.

This function posts a scheduled action request for a GETFTP generation.

When the function returns, the *GETSYS PRG, "ACTIONID"* returns the ID of the scheduled action and allows tracking this action. It is also possible to program an ONSTATUS action that will be called when the action is finished (with or without success).

Example 57: Use of GETFTP

```
a$ = "source-file-name.txt"
b$ = /usr/destination-file-name.txt

// Transfer a file
GETFTP a$, b$

// Transfer a file with address + credentials
c$ = "user:pwd@ServerTP.com:21, 1"
GETFTP a$, b$, c$

// Append the content to a root document
GETFTP "inst_val.txt", "/inst_val.txt"
```

Check also

ONSTATUS, p. 78; *GETSYS, SETSYS, p. 44*; *PUT, p. 88*

3.30 GETHTTP

Syntax

```
GETHTTP S1, S2, S3[, S4]
```

S1	the connexion Parameter formatted as: *[user:password@]servername[:port]*
S2	the file name to assign on the device eWON formatted as: file name path
S3	the URI of the file on the HTTP formatted as: server absolute path of the file to be downloaded
S4	"PROXY"

Description

The *GETHTTP* command submits an HTTP GET request. It allows the download of a file (one per *GETHTTP* command) using its URI.

When the function returns, the *GETSYS PRG*, returns the ID of the scheduled action and allows the tracking of this action. It is also possible to program an *ONSTATUS* action that will be called when the action is finished (with or without success).

When *"PROXY"* is added at the end of the command, the device performs the *GETHTTP* through a proxy server. The device uses the proxy server parameters configured in the Internet connection proxy parameters on the VPN Global section of the web interface.

By default, when no port is provided, the HTTP port is 80.

Example 58: Perform a GETHTTP Request

```
b$ = "/usr/filename1.txt"
c$ = "/filename1.txt"

// Download without HTTP basic authentication
a$ = "10.0.100.206"
GETHTTP a$, b$, c$

// Download with basic authentication and configured HTTP port
a$ = "adm1:adm2@www.ewon.biz:89"
GETHTTP a$, b$, c$

// Download without HTTP basic authentication through proxy serveur
GETHTTP a$, b$, c$, "PROXY"
```

Check also

ONSTATUS, p. 78; *GETSYS, SETSYS, p. 44*; *PUTHTTP, p. 91*

3.31 GETIO

Syntax

```
GETIO S1
```

S1	the tag reference (tag name, ID or index)

Description

This function returns the value of the *S1* tag set in the device. The value of this tag is a float.

This function is equivalent to *a = MyTag@*. The *MyTag@* BASIC variable is distinct than the device memory tag “MyTag”.

Example 59: Get the Value of a Tag

```
a = GETIO "MyTag"
a = GETIO 12  // Valid only if there’s a tag ID = 12
```

Check also

Tag Access, p. 21

3.32 GETSYS, SETSYS

The *GETSYS* and *SETSYS* functions are used to get or set some special parameters of the device.

There are 5 types of parameters:

GETSYS / SETSYS Parameters Types	
Group	**Description**
PRG	Program parameters such as the time in milliseconds or the type of action that started the program
SYS	Modification of the device system parameters
COM	Modification of the device communication parameters
USER	Modification of the device users list
TAG	Modification of the device tag list
INF	Information about the device (debug counter,...)

Each group has a number of fields that can be read of written.

3.32.1 Procedure

The procedure is the same for each group call:

1. A block must be loaded for modification with the *SETSYS* command and a special field called *"load"*.

   ```
   SETSYS TAG, "load", XXXXXXX
   ```

   According to the source, this block will be either the device system configuration, the device COM configuration, a tag configuration or a user configuration.

2. Each field of this configuration can be accessed by the *GETSYS* or *SETSYS* commands. This modification works on the record loaded values but does not actually affect the configuration.

3. When the modifications are done, the *SETSYS* command is called with a special field called *"save"* and the edited block is saved. This is only necessary if the record has changed.

   At this time, the record edited content is checked, the configuration is updated and applied.

4. The *CFGSAVE* command can be called to save the updated configuration to flash.

Recognized Field Values per Group

The fields values are the same fields as those returned by the “GET config.txt” command through FTP.

Syntax

```
GETSYS SSS, S1
```

SSS	the source block: *PRG*, *SYS*, *TAG*, *USR*. This parameter must be typed as is, it could not be replaced by a string!
S1	the field name that needs to be read or modified

Syntax

```
SETSYS SSS, S1, S2|E2
```

SSS	the source block: *PRG*, *SYS*, *TAG*, *USR*. This parameter must be typed as is, it could not be replaced by a string!
S1	the field name that needs to be read or modified. It can also be the action *"load"* or *"save"*
S2|E2	the value to assign to the field. The type of the value depends on the field itself.

Example 60: GETSYS & SETSYS

```
a% = GETSYS PRG, "TIMESEC"
// Supposedly Tag_1 exists and is a memory tag
SETSYS TAG, "load", "Tag_1"
// a$ = "Tag_1"
a$ = GETSYS TAG, "Name"
// EmailTo field of Tag_1
SETSYS TAG, "ETO", "ewon_actl@ewon.biz"
// save data in the config which results in the update of Tag_1
SETSYS TAG, "save"
SETSYS TAG, "Name", "Tag_2"
// Update or create Tag_2 with Tag_1 config
SETSYS TAG, "save"
```

Check also

CFGSAVE, p. 25;

TAG Load

The *TAG load* case is particular because it allows the load of a tag defined by its name, ID or index.

If there are 6 tags defined in the config, each tag can be accessed by its name, its index (0 to 5) or its ID. The ID is the first item of a tag definition when reloading the config.txt file. It is never reused until the device is formatted (with a reset level 2).

Method	XXX Param.	Example	Explanation
Tag name access	Tag name	*SETSYS TAG, "load", "MyTagName"*	Loads tag named “MyTagName”
Index access	Index	*SETSYS TAG, "load", - 4*	Loads Tag which index is 4
TagId access	Id	*SETSYS TAG, "load", 50*	Loads Tag which id is 50

Check also

Tag Access, p. 21

Extended Syntax to Access IO Server Lists of Parameters

General Syntax

```
GETSYS SYS, "ParamName:SubParamName"
SETSYS SYS, "ParamName:SubParamName", "NewValue"
```

ParamName	the name of the whole field form the config.txt file.
SubParamName	the sub-parameter (inside the ParamName) that needs to be read or modified.
NewValue	the value to assign to the field.

Specific IOServer Syntax

```
GETSYS SYS, "IOSrvData[IOServerName]:SubParamName"
SETSYS SYS, "IOSrvData[IOServerName]:SubParamName", "NewValue"
```

IOServerName	the name of the IOServer you want to edit form the config.txt file).
SubParamName	the sub-parameter (inside the IOSrvData[...]) that needs to be read or modified.
NewValue	the value to assign to the field.

Description

These commands allow an easy access to sub-parameters contained in a parameter string.

Example 61: IO Server

```
// Generic syntax
SETSYS SYS, "load"
A$ = GETSYS SYS, "IOSrvData2:GlobAddrA"
SETSYS SYS, "IOSrvData2:GlobAddrA", "0,254,0"

// Specific IO server syntax
SETSYS SYS, "load"
A$ = GETSYS SYS, "IOSrvData[UNITE]:GlobAddrA"
SETSYS SYS, "IOSrvData[UNITE]:GlobAddrA", "0,254,0"
```

3.32.2 Parameter Type: PRG

ACTIONID

Operation Read | Write

Type Integer

Description After execution of a scheduled action such as *SENDSMS*, *SENDMAIL*, *PUTFTP*, *SENDTRAP* or*TCP/UDP Connect*, the *ACTIONID* returns the ID of the action that had just been executed.

When the *ONACTION* event is executed, this *ACTIONID* is stored in EVTINFO. Writing in this field is useful to read the current value of an action.

ACTIONSTAT

Operation Read only

Type Integer

Description Current status of the action with ActionID given by ACTIONID.

If ACTIONSTAT needs to be checked, ACTIONID must be initialized first.

Possible values of ACTIONSTAT are:

- -1: in progress
- -2: ID not found
- 0: done with success
- >0: finished with error. The number is the error code

The device maintains a status list of the last 20 scheduled actions that were executed. When more actions are executed, the older status is erased and its ACTIONSTAT may return –2, meaning it is not available anymore.

ADSLRST

Operation Write

Type Integer

Description Force a hardware ADSL modem reset: *SETSYS PRG, "ADSLRST", 1*

EVTINFO

Operation Read only

Type Integer

Description The value of this field is updated before executing the *ONXXXXX* (*ONSTATUS*, *ONERROR*, etc.).

Check the different *ONXXXXX* function to learn the meaning of the EVTINFO parameter.

LSTERR

Operation	Read | Write
Type	Integer
Description	Contains the code from the last BASIC error that occurred. If *-1* is returned, this means no error was found. Check the *BASIC Error Codes, p. 108* The *LSTERR* is automatically cleared (value -1) when an end of section is reached (instruction END). LSTERR can be cleared by setting the parameter to -1: *SETSYS PRG, "LSTERR", -1*.

MDMRST

Operation	Write
Type	Integer
Description	Force an hardware modem reset: *SETSYS PRG, "MDMRST", 1*

MSEC

Operation	Read only
Type	Integer
Description	Time in MSEC since the device has booted. Maximum value is 134217727, afterwards it drops to 0.

NBTAGS

Operation	Read only
Type	Integer
Description	*NBTAGS* returns the number of tags defined in the device.

PPPIP

Operation	Read | Write
Type	String | Integer
Description	This parameter returns the string corresponding to the current PPP IP address. When the device is offline, the returned value is “0.0.0.0”. When the device is online, the returned value is the dotted IP address allocated for the PPP connection. The parameter can be written to disconnect the device. The only accepted value when writing in this parameter is 0: *SETSYS PRG, "PPPIP", 0*

PRIOH

Operation	Read | Write
Type	Integer
Description	Used to change the script priority

PRION

Operation	Read | Write
Type	Integer
Description	Used to change the script priority

PRIOT

Operation	Read | Write
Type	Integer
Description	Used to change the script priority

RESUMENEXT

Operation	Read | Write
Type	Integer
Description	This parameter controls the *ONERROR* action. Possible values are a combination of: • 1: Resume next mechanism is enabled • 4: Do not execute ONERROR • 8: Do not show error on virtual screen This parameter is useful when testing the existence of a variable, file or other. Example: Testing the existence of a file can be done by opening it and see if it generated an error. The error result is accessible through *LSTERR*

RUNSRC

Operation	Read only
Type	Integer
Description	When program is started, the source of the execution is given by this parameter: • 1: Started from the web interface 'Script Control' window • 2: Started by the FTP server because program has been updated • 3: A 'GO' command has been executed from the script • 4: Automatic program starts at the boot of the device

SCHRST

Operation	Write
Type	Integer
Description	Clear all pending scheduled actions (except the action currently “in progress”). Write only with the value *1*: *SETSYS PRG, "SCHRST", 1* When Scheduled Actions are cleared, they have the status “Action Canceled” which value is 21613.

SERNUM

Operation	Read | Write
Type	String
Description	This parameter returns a string with the device serial number string.

TIMESEC

Operation	Read only
Type	Integer
Description	This parameter returns the time elapsed since 1970-01-01 in seconds which can be useful to compute time differences. When this value is assigned to a float variable, the number is too big and rounding will occur. To store this value, an integer variable should be used instead (i.e: a%) .

TRFWD

Operation	Read | Write
Type	String
Description	Transparent forwarding IP address. The parameter can be used to write or read the routing parameter. It is only active when the PPP connection is established.

VPNIP

Operation	Read only
Type	String
Description	Currently allocated VPN IP address. If the device is not connected to VPN, the value is “0.0.0.0”

WANIP	
Operation	Read only
Type	String
Description	This parameter returns the string corresponding to the current WAN IP address. Depending on the device configuration, it can be the WAN Ethernet, Wi-Fi or the modem connection. When the device is offline, the returned value is “0.0.0.0”. If the device has performed a dynDNS request or an Internet connection check (using the Internet connection wizard for example), then the *WANIP* parameter will reflect the public IP address used for the Internet connection. For example, if the device connects to Internet using a router, then the *WANIP* parameter will reflect the public IP address used by this router.

3.32.3 Parameter Type: SYS

The fields edited within this group are the ones found in the config.txt file under the System section on the web interface of the device.

Check also

Configuration Fields, p. 109

3.32.4 Parameter Type: COM

The fields edited within this group are the ones found in the comcfg.txt. It is also possible to tune the modem detection.

Check also

Configuration Fields, p. 109

3.32.5 Parameter Type: INF

This group holds all information data about the device. All these fields are read only. The fields displayed from this group are the ones found in the estat.htm file.

3.32.6 Parameter Type: TAG

The fields edited within this group are the ones found in the config.txt file under the section TagList.

Check also

Configuration Fields, p. 109

3.32.7 USER

The fields edited within this group are the ones found in the config.txt file under the section “UserList”.

Check also

Configuration Fields, p. 109

3.33 GO

Syntax

```
GO
```

Description

This command starts the execution of the program. This is equivalent to clicking “RUN” in the BASIC IDE window.

This command is mainly useful for remote device operation through the use of “remote.bas” FTP transfer.

3.34 GOSUB - RETURN

Syntax

```
GOSUB S1
S1:
...
RETURN
```

S1	the name of a label.

Description

When the *GOSUB* line is executed, the program continues but jumping to *Label* line. The program executes the code until the *RETURN* line is met. The *RETURN* command modifies the program pointer to the line immediately following the *GOSUB* Line.

It is possible to create a new section containing the *Label*. Sections are useful in order to divide the program into smaller code snippets and help the reader to get a clear view of the software.

At the end of every section there is an invisible *END* but jumps are possible from section to section.

Example 62: Use of the GOSUB

```
GOSUB NL3
PRINT "End"
END
NL3: PRINT "Beginning"
RETURN  // Prints "Beginning" then "End"

GOSUB NL3 : PRINT "Never"
PRINT "End"
END
NL3: PRINT "Beginning"
RETURN  // Prints "Beginning" then "Never" then "End"
```

3.35 GOTO

Syntax

```
GOTO S1
```

S1 the name of the label.

Description

The execution of the program jumps to the *label* line. The *label* statement cannot be empty.

The *GOTO* command also allows starting the program without erasing all variables.

A string variable can be passed in a *GOTO* command.

Example 63: Jump to a Specific Label

```
GOTO MyLabel
PRINT "Hop"  // "Hop" is never printed
Label:
...

a$ = "my_label"
GOTO a$
PRINT "Hop"  // "Hop" is never printed
my_label:
...
```

3.36 HALT

Syntax

```
HALT
```

Description

This command stops the execution of the program. This is similar to clicking *STOP* in the BASIC IDE window. This command is mainly useful for remote device operation through the use of “remote.bas” FTP transfer.

Check also

GO, p. 52; *REBOOT, p. 92*

3.37 HEX$

Syntax

```
HEX$ E1
```

Description

The function returns a chain of 8 characters that represents the hexadecimal value of the *E1* number.

Example 64: Convert from DEC to HEX

```
a$= HEX$ 255  // a$ = 000000FF
```

Check also

BIN$, p. 24

3.38 HTTPX

The BASIC implemented in the device is capable of dealing with HTTP(S) request & response.

3.38.1 REQUESTHTTPX

Syntax

```
REQUESTHTTPX http[s]://S1, S2[, S3[, S4[, S5[, S6[, S7]]]]]
```

S1 the server.

It is the URL of the targeted request. For example: “192.168.0.10” or “www.example.com”.

It is also part of the URL that constitute the query string. For example: “service” or “12345/control?axis=x&val=1”

S2 the method.

It's the REST API HTTP verb. This can be “GET”, “POST”, “PUT”, “PATCH”, “DELETE”, “OPTIONS”, “HEAD” or “PURGE”.

This parameter must be set in upper case (some servers don’t allow methods to be sent in lower case).

S3 the headers.

The headers sent by the device through the request. For example: “ContentType= application/json&XRequest =test”

S4 the post data.

The POST data can be separated either by an ampersand “&” using the traditional querystring format *[FieldName1=ValueName1] [&FieldNameX=ValueNameX]*. For example: “firstname=jack&lastname=nicholson”.

Or it can be separated by raw data. For example: “{\"myData\":21}”.

S5 the file data.

String for FILE data separated by an ampersand “&” using the traditional querystring format and having each value corresponding to an Export Block Descriptor: *[FieldName1=ExportBlockDescriptor1]*. For example: “pictures[]=[$dtEV$fnevents.txt]&pictures[]=[$dtCF$fnconfig.txt]”.

S6 the file answer.

The file name inside */usr/* folder where the answer needs to be stored. For example: “/usr/myfile”.

S7 the proxy.

This option indicates if the request should use a proxy or not. Accepts "PROXY" or " " as value.

Specifications

When the *file-answer* field is empty or not specified:

- the result of the request is saved in a buffer inside the memory. The information can then be retrieved with the *RESPONSEHTTPX* command.
- there are three buffers: each buffer can handle a response body of max. 64KB. An HTTP request error is produced if the response body is bigger than the max. size allowed.

Whenever the *post-data* field is specified without any *file-data* information, the default content type header (if not specified in the *header* field) is

```
'Content-Type: application/x-www-form-urlencoded; charset=ISO-8859-1'
```

If you try to send a file, the header **<Content-Type>** is automatically set to **application/x-www-form-urlencoded**. If you try to override this header, the file will not be sent.

When using a “multipart/form-data” content type, it is not possible to set the boundary.

```
// Not supported
'Content-Type:multipart/form-data; boundary=--------myseparator'
```

3.38.2 RESPONSEHTTPX

Syntax

```
RESPONSEHTTPX S1[, S2]
```

S1	the parameter. It is sending the info to retrieve. For example: “HEADER”, “STATUSCODE” or “RESPONSE-BODY”
S2	the specific header. If a specific header need to be retrieved. This works only when *S1* is set to “HEADER”.

Specifications

RESPONSEHTTPX is used to retrieve the information from a previous *REQUESTHTTPX* command.

Use the *ACTIONID* (parameter from the *GETSYS PRG*) to specify the request:

RESPONSEHTTPX "HEADER"	it returns all server headers with the format “HeaderName: value” separated by CR+LF (ascii 13dec then 10dec).
RESPONSEHTTPX "HEADER", "Specific-Header"	it returns only “Specific-Header: value” or an empty string if not found.
RESPONSEHTTPX "STATUSCODE"	it returns the request status code (“200”, “404”...) as a string.
RESPONSEHTTPX "RESPONSE-BODY"	it returns the response body as a string that can contain NULL characters.

Example 65: Use of RESPONSEHTTPX

```
request:
ONSTATUS “GOTO onEvent”
REQUESTHTTPX "http://www.example.com/hello.php","GET"
actionID% = GETSYS PRG, "ACTIONID"
PRINT "request actionid is "; actionID%
END

onEvent:
eventId% = GETSYS PRG, "EVTINFO"
IF (eventId% = actionID%) THEN
  SETSYS PRG, "ACTIONID", eventId%
  stat% = GETSYS PRG, "ACTIONSTAT"
  IF (stat% = 0) THEN
    GOTO response
```

```
  ELSE
    PRINT "Error (ERROR = "+Str$(stat%) + ")"
  ENDIF
ENDIF
END

response:
a$ = RESPONSEHTTPX "STATUSCODE"
PRINT "status: "; a$
a$ = RESPONSEHTTPX "HEADER"
PRINT "all headers: "; a$
a$ = RESPONSEHTTPX "HEADER", "Server"
PRINT "server header: "; a$
a$ = RESPONSEHTTPX "RESPONSE-BODY"
IF (Len(a$) < 1000) THEN
  PRINT "response body: "; a$
Else
  PRINT "response body size: "; Len(a$)
ENDIF
END
```

3.39 IF, THEN, ELSE, ENDIF

This sequence of commands supports two different syntaxes: the short and long IF syntax.

3.39.1 Short Syntax

Syntax

```
IF N THEN EXPRESSION1[ ELSE EXPRESSION2][ ENDIF]
```

Description

The condition is the result of an operation returning an *N* integer.

- If *N* is 0, the condition is considered as false and the device executes the next line or the *ELSE EXPRESSION2* if available.
- If *N* is different than 0, the condition is considered as true and the device executes *EXPRESSION1*.

If more than one instruction have to be executed, separate them with a colon ":".

If *N* is an expression or a test, use parenthesis ().

The short *IF* syntax is used as soon as an item is found after the *THEN* statement. Even putting a comment statement on the *IF N THEN* line will make the device consider it as a short *IF* statement.

If *ELSE EXPRESSION2* is expressed, then the *ENDIF* statement is mandatory.

Example 66: Short IF Syntax

```
IF (a < 10) THEN PRINT "a is lower than 10" : SETIO "MyTag", 1
```

3.39.2 Long Syntax

Syntax

```
IF N THEN
  EXPRESSION1[
ELSE
```

```
  EXPRESSION2]
ENDIF
```

Description

Short and long *IF* syntax can be mixed in the code but anything typed after the *THEN* statement will lead to a short *IF* syntax interpretation.

Example 67: Long IF Syntax

```
IF (a < 10) THEN
  PRINT "a is lower than 10" : MyTag@ = 1
ELSE
  PRINT "a is bigger than 10" : MyTag@ = 0
ENDIF
```

3.40 INSTR

Syntax

```
INSTR I1, S1, S2
```

I1	the index in the string to search. Valid value goes from 1 to LEN *S1*).
S1	the string that will be searched in.
S2	the string to search for in *S1*

Description

The function returns an integer equal to the position of string *S2* in string *S1*.

- If string *S2* is found, the function returns a value from 1 to the length of *S1*. The returned index is 1 based.
- If string *S2* is not contained in *S1*, the function returns 0.

The *I1* parameter should be 1 to search the whole *S1* string.

If *I1* is higher than 0 then string *S1* is searched starting at offset *I1*. The value returned is still based on *S1* offset.

Internally, the *INSTR* function uses the character “0” (0x00) as delimiter. This means that the character “0” can not be searched with “INSTR”. The result will always be 1 even if there is no “0” in the searched string.

Example 68: Find a String Inside a String

```
INSTR 1, "AAABBC","BB"  // Returns 4
INSTR 3, "AAABBC","BB"  // Returns 4

B$ = CHR$(0)
A% = INSTR 1, A$, B$  // Always returns 1
```

3.41 INT

Syntax

```
INT F1
```

Description

Extract the integer part out of the number. There is no rounding operation.

Example 69: Extract an Integer

```
a = INT(10.95)  // a equals 10.00, still float
A% = 10.95  // a equals 10, automatic type conversation
```

3.42 IOMOD

Syntax

```
IOMOD S1
```

S1	the tag reference (tag name, ID or index)

Description

This function returns '1' if the *S1* tag value has been modified in the device since the last call of the *IOMOD* command.

The call to this function resets the internal change tag flag to 0. It the variable doesn't change anymore, the next call to *IOMOD* will return 0. A similar behavior can be achieved with the use of *ONCHANGE* event handler.

Example 70: Get notified when Tag Has Changed

```
a% = IOMOD "MYTAG"
IF a% THEN PRINT "mytag has changed"
```

Check also

ONCHANGE, p. 73

3.43 IORCV

Syntax

```
IORCV S1[, I1]
```

S1	the IOServerName parameter.
I1	an optional additional parameter. Can be 0, 1, -1.

Description

The *IOSEND* and *IORCV* functions must be used together. They are used to send/receive custom IO server requests.

These functions can be used only if IO server is configured. Use *IORCV* function to read the IO server response from an *IOSEND* request.

There are three transmission slots available. Using *IORCV* allows the clearing of those before the three slots are busy. Requests are interlaced between gateway requests sent to the IO server and normal IO server polling operations.

- **First Case**

```
a$ = IORCV a%
a$ = IORCV a%, 0
```

This first case returns the result or the status of the request.

a% holds the request number and is the result of the *IOSEND* command. The possible returned values:

a$ = "XXXXXXXXX"	where "XXXXXXXXX" the result of the request
a$ = "#FREE"	slot a% is free
a$ = "#RUN"	slot a% is in progress
a$ = "#ERR"	slot a% is done with error

If the request is done (all cases except *"#RUN"*), the slot is always freed after the *IORCV a%* or *IORCV a%, 0*.

- **Second Case**

```
a$ = IORCV a%, -1
```

This is the same as the first case *a$=IORCV a%, 0* except the slot is not freed if a request is done.

- **Third Case**

```
b% = IORCV a%, 1
```

This returns the status of the *IORCV* command as an integer. The slot is not freed by this parameter.

The returned status can contain the following values:

b% = -2	slot a% is free.
b% = -1	slot a% is in progress.
b% = 0	slot a% is done with success.
b% > 0	slot a% is done with error.
b% < -2	lot a% is done with error – code type: warning.

In the situation where *b% < 2*, such warning codes mean “Read failed” on the serial link. These warnings are flagged as internal and thus are not added in the event log. Those codes can be very long; ie. -536893114

Example 71: IORCV

```
TestIO:
// The following creates the Modbus command
a$ = chr$(4) + chr$(0) + chr$(0) + chr$(0) + chr$(1)
// Initiate the Modbus request on slave 21
a% = IOSEND "MODBUS", "21", a$

Wait_IO_End:
b% = IORCV a%, 1  // read the status
IF b% = -1 THEN
  GOTO Wait_IO_End  // If idle then loop
ENDIF
b$ = IORCV a%  // Read the result and free the slot
PRINT LEN(b$)
PRINT b$
END
```

3.44 IOSEND

Syntax

```
IOSEND S1, S2, S3
```

S1	the IO server name as it appears in the tag configuration page.
S2	the slave address as described in the device user manual for each IO server section.
S3	the array of bytes with a protocol command, the content depends on the IO server.

Description

This function Returns a request number (slot) that must be used in *IORCV* to read the response of the request.

The request result is read by using the *IORCV* function and uses a polling mechanism. It means *IORCV* should be used to check via the request received through *IOSEND* that the slot is free.

There are three transmission slots available. Using *IORCV* allows the clearing of those before the three slots are busy. Requests are interlaced between gateway requests sent to the IO server and normal IO server polling operations.

Example 72: Get the Value of a Tag

```
a% = IOSEND IOServerName, Address, IoCommand
```

Check also

IORCV, p. 59

3.45 LEN

Syntax

```
LEN S1
```

S1	the string which length will be calculated

Description

This function returns the number of characters of a string.

Example 73: Calculate the number of characters

```
a$= "Hop "
a% = LEN a$
PRINT a%  // Prints 4
```

3.46 LOGEVENT

Syntax

```
LOGEVENT S1[, S2]
```

S1	the string to log.
S2	the type of logging

Description

This command appends an event to the log file. The current time is automatically used for event logging.

The *S2* can take different ranges of value:

Range of values	Description
0 … 99	Error
-99 … -1	Warning
100 … 199	Trace

Example 74: Log an event

```
logevent "Save this in log", 120
// Entry log: 978353046;"01/01/2001 12:44:06";"Save this in log"
```

3.47 LOGGROUPIO

Syntax

```
LOGGROUPIO S1[, E1[, E2]]
```

S1	The group(s) of tags that should be logged. Possible values: A, B, C and/or D
E1	The rounding time defines the time increment used to record the tag values. Default value: 1 Possible values: 2, 3, 4, 5, 6, 10, 12, 15, 20, 30 or 60
E2	The time rounding method Default value: 1 Possible values: 0 (nearest), 1 (truncate)

Description

This command logs all tags belonging to the specified group list *S1* using the same timestamp for the recording.

Before being able to force it, the tag must have historical logging enabled.

The Historical Logging will record times that are multiples of the *E1* parameter.

E2 selects the method used to round the time:

- *0* : “nearest” will record the time as the nearest rounding time increment defined. Example: LogGroupIO “A”,20,0 a tag value sampled at 10:34:16 will be recorded as 10:34:20 - 1 : “truncate” will record the time as the last RoundingTime increment defined. Example: LogGroupIO “A”,20,1 a tag value sampled at 10:34:16 will be recorded as 10:34:00

Example 75: Record Group of Tags under the same Timestamp

```
// Log Group A each second
LOGGROUPIO "A"

// Log all groups each 20 seconds
LOGGROUPIO "ABCD", 20

// A tag sampled at 10:34:16 will be logged as 10:34:20
LOGGROUPIO "A", 20, 0

// A tag sampled at 10:34:16 will be logged as 10:34:00
LOGGROUPIO "A", 20, 1
```

3.48 LOGIO

Syntax

```
LOGIO S1
```

S1	the tag reference (tag name, ID or index)

Description

This commands forces the historical logging of *S1* tag.

Before being able to force it, the tag must have historical logging enabled.

The point is logged at the time the *LOGIO* command is issued with its current value.

If the tag is configured for historical logging with logging dead band equal to –1 and time interval equal to 0, no value will be logged automatically and it is possible to program a pure scripted logging.

Example 76: Log a Tag Value by Script

```
LOGIO "mytag"
```

3.49 LTRIM

Syntax

```
LTRIM S1
```

Description

LTRIM returns a copy of a string with the leftmost spaces removed.

Example 77: TRIM a string

```
b$ = LTRIM a$
```

3.50 MEMORY

Syntax

```
MEMORY S1
```

S1	one of the 3 following values: *"PROG"*, *"VAR"*, *"TOT"*

Description

Depending on the value of *S1*, it will return the free memory of a specific zone:

- *"PROG"* returns the free memory of the program zone.
- *"VAR"* returns the free memory of the variable zone.
- *"TOT"* returns the free memory of *"PROG"* + *"VAR"*

3.51 MOD

Syntax

```
E1 MOD E2
```

Description

This computes the remainder of the division of *E1* by *E2*.

Example 78: Modulo Operation

```
1 MOD 2  // Returns 1
2 MOD 2  // Returns 0
```

Check also

Operators Priority, p. 17

3.52 MONTH

Syntax

```
MONTH E1|S1
```

E1	a date in integer format which represent the number of seconds since 1970-01-01
S1	a date in string format under the form: DD/MM[/YYYY HH:MM:SS] (i.e: "18/09/2003 15:45:30").

Description

This function returns an integer corresponding to the value of the month (1 — 12) that matches a defined time variable.

Do not call the function with a float variable of value or this would result to error “invalid parameter”.

Example 79: Retrieve the Month

```
a$ = TIME$
a% = MONTH a$

b% = GETSYS PRG, "TIMESEC"
a% = MONTH b%
```

Check also

DAY, p. 26; *DOW, p. 28*; *DOY, p. 28*; *WOY, p. 104*

3.53 MQTT

The BASIC provides a MQTT API allowing one MQTT client.

This client is asynchronous (based on events in background) and supports:

- MQTT protocol: version 3.1 and 3.1.1
- MQTT TLS encryption: TSL 1, TLS 1.1 and TLS 1.2
- MQTT client authentication:
  - user and password.
  - certificate and private key.
- MQTT server authentication using Certificate Authority.

Syntax

```
MQTT "CMD"[, PARAMS]
```

Description

This is the main command to configure and manipulate the MQTT client

All possible commands for the MQTT are listed hereunder.

Example 80: MQTT Publishing & Subscribing

```
Onwan '@WANAction(GETSYS PRG, "EVTINFO")'

Function WANAction($WANStatus%)
  IF $WANStatus% = 1 Then
    PRINT "WAN up"
    @Start()
  ENDIF
ENDFN

Function Start()
  PRINT "starting MQTT"
  @MosquittoInit()
  @MosquittoConnect()
ENDFN

Function MosquittoInit()
  MQTT "open", "ewon_flexy", "test.mosquitto.org"
  MQTT "setparam", "log", "1"
  MQTT "subscribe", "ewons/test/messages", 0
ENDFN

Function readMsg($msgID%)
  IF $msgID% > 0 Then
    msgTopic$ = MQTT "msgtopic"
    msgData$  = MQTT "msgdata"
    PRINT "received: "; msgTopic$; " -> "; msgData$
    @readMsg(Mqtt("read"))
  ENDIF
ENDFN

Function MosquittoConnect()
  MQTT "connect"
  ONMQTT '@readMsg(mqtt("read"))'
  ONMQTTSTATUS '@MosquittoMQTTStatusChange(mqtt("status"))'
ENDFN

Function MosquittoMQTTStatusChange($status%)
  IF $status% = 5 Then
    PRINT "MQTT connected"
    TSET 1, 5
    ONTIMER 1, "@MosquittoMQTTPublish()"
  ELSE
    PRINT "MQTT disconnected"
  ENDIF
ENDFN

Function MosquittoMQTTPublish()
  msg$ = "test(" + STR$(n%) + ")"
  MQTT "publish", "ewons/test/messages", msg$, 0, 0
  n% = n% + 1
ENDFN

n% = 1
```

3.53.1 OPEN

Syntax

```
MQTT "open", S1, S2
```

S1	the client ID
S2	the broker host

Description

This command opens an MQTT connection to a broker *S2* and register itself to this broker using the ID *S1*.

Example 81: MQTT OPEN API

```
MQTT "open", "ewon_flexy", "test.mosquitto.org"
```

3.53.2 SETPARAM

Syntax

```
MQTT "setparam", S1, S2
```

S1	the name of the parameter
S2	the value of the parameter

Description

Parameters can be set through this “setparam” request.

The “open” request must be called before using the “setparam” request.

Available Parameters	
port	The TCP port of the MQTT broker. Default: 1883 (which is usually used for unencrypted MQTT). Values: between 0 and 65535
username	The username used to log in.
password	The password used to log in.
keepalive	The number of second between two MQTT heartbeats. Default: 60 Values: between 0 and 65535
CAFile	The path to the certificate authority file, used in PEM format. For example: /usr/root-ca.pem
CAPath	The path to a directory containing the certificates. For example: /usr/certs/
CertFile	The path to the client certificate, used in PEM format. For example: /usr/flexy.crt.pem
KeyFile	The path to the client private key, used in PEM format. For example: /usr/flexy.private.key
CleanSession	Indicates if the client session information (subscription, etc.) on the broker should be stored between a disconnect/reconnect. Not all servers allows you to set this flag to false. Default: 1 (session is cleaned when disconnecting). Values: 0 or 1
ProtocolVersion	Choose the MQTT protocol version. Some servers only implements a single version leaving beside the others. Default: 3.1

	Values: 3.1.1
TLSVersion	Force the TLS version to use. Some servers only support a single type of TLS and block the others. Default: tlsv1.2 Values: tlsv1, tlsv1.1 or tlsv1.2
Log	Allows the printing of the client verbose logs in the Realtime Log event of the device. Default: 0 Values: 0 or 1
MaxInFlight	Allows the control of the maximum number of messages retained on the client side. Default: 20 Values: between 0 and 50
WillPayLoad	The message stored by the MQTT broker and sent out to subscribers if the Ewon disconnects unexpectedly.
WillTopic	The topic associated to the Will message
WillQoS	The QoS to use for the Will message. Default: 0 Values: 0,1,2
WillRetain	If set, the will message will be processed as a retained message. Default: 0 Values: 0 or 1

Example 82: MQTT OPEN API

```
MQTT "setparam", "port", "1883"
```

3.53.3 CONNECT

Syntax

```
MQTT "connect"
```

Description

This command starts the connection with the broker.

The “connect” request must be called after the “open” and the “setparam” requests have both been called.

The “connect” call is asynchronous, it is non blocking and is executed in the background. Time to connect may vary depending on the device usage, the broker and the Internet connection.

3.53.4 CLOSE

Syntax

```
MQTT "close"
```

Description

This command stops the connection with the broker and removes all the subscriptions.

To use this call, the client should be connected. Check can be done by using *MQTT "status"*.

The “close” call is asynchronous, it is non blocking and is executed in the background. Time to connect may vary depending on the device usage, the broker and the Internet connection.

3.53.5 STATUS

Syntax

```
MQTT "status"
```

Description

This command allows the retrieval of the current MQTT client status.

Possible values are:

3	The MQTT client is trying to connect.
4	The MQTT client is disconnected.
5	The MQTT client is connected.

Connection process can last a little (depending on the broker, Internet connection...). If errors occur, they will appear during this connection attempt stage. If it is the first time the setup is being configured, check the event log or real time log of the device for more information.

Example 83: Status of the MQTT Communication

```
status% = MQTT "status"
```

3.53.6 PUBLISH

Syntax

```
MQTT "publish", S1, S2, E1, E2
```

S1	the topic
S2	the message
E1	the quality of service (QoS)
E2	the retained message feature.

Description

This call is asynchronous, it is non blocking and is executed in the background. Time to connect may vary depending on the device usage, the broker and the Internet connection.

The QoS and retained messages feature rely on the broker capabilities. In this matter, make sure that it is supported. If unsure, *0* should be used for both.

Quality of Service Values	
0	Message is delivered without any confirmation, the message could be lost.
1	Message has been delivered at least once, the server acknowledges each message to the client.
2	Message has been delivered only once, 4-way handshake between client/server to ensure the message is correctly delivered.

Retained Message Feature Values	
0	Message should not be retained on broker side.
1	The broker should keep messages even after sending it to all current subscribers. If a new subscription is made on the topic, this new subscriber will receive the retained messages.

Example 84: Publish a Message through MQTT

```
MQTT "publish", "ewons/alarms", "test message", 0, 0
```

3.53.7 SUBSCRIBE

Syntax

```
MQTT "subscribe", S1, E1
```

S1	the topic
E1	the quality of service (QoS)

Description

This call is asynchronous, it is non blocking and is executed in the background. Time to connect may vary depending on the device usage, the broker and the Internet connection.

Wildcards can be used in *S1*:

+	used to wildcard a topic level	
	“a/b/+/d”	matches “a/b/c/d”, “a/b/c1/d”, “a/b/tags/d”... doesn’t match “a/b/c”
	“+/+/+/+”	matches “a/b/c/d”, “d/a/c/b”... doesn’t match “a/b/c”
#	used to wildcard the suffix	
	“a/b/#”	matches “a/b/c”, “a/b/c/d”... doesn’t match “a/c”
	“#”	matches anything

Topic levels can be of length 0 such as “a//topic” (topic level 2 is an empty string) and can lead to non obvious wildcard matches or subscription behavior.

The QoS relies on the broker capabilities. In this matter, make sure that it is supported. If unsure, *0* should be used.

Quality of Service Values	
0	Message is delivered without any confirmation, the message could be lost.
1	Message has been delivered at least once, the server acknowledges each message to the client.
2	Message has been delivered only once, 4-way handshake between client/server to ensure the message is correctly delivered.

Example 85: Subscribe to a Topic through MQTT

```
MQTT "subscribe", "ewons/alarms", 0
```

3.53.8 READ

Syntax

```
MQTT "read"
```

Description

This function returns the ID of the oldest unread message received by the MQTT client (based on the FIFO system).

If the returned value is 0, it means that you don’t have any message.

- *0*: no message available
- *>0*: message is out of the queue and available through “msgtopic” and “msgdata” calls.

To empty the client queue message, *MQTT "read"* must be called until 0 is returned.

Example 86: Retrieve the Oldest Unread Message

```
msgID% = MQTT "read"
```

3.53.9 MSGTOPIC

Syntax

```
MQTT "msgtopic"
```

Description

This command gives you the topic of the read message.

Example 87: Retrieve the Topic of a Message

```
msgTopic$ = MQTT "msgtopic"
```

Check also

READ, p. 70

3.53.10 MSGDATA

Syntax

```
MQTT "msgdata"
```

Description

This command gives you the data (payload) of the read message.

Example 88: Retrieve the data of a Message

```
msgData$ = MQTT "msgdata"
```

Check also

READ, p. 70

3.54 NOT

Syntax

```
NOT E1
```

Description

This function returns “1” if *E1* is equal to “0” otherwise the function returns “0”.

Example 89: Negation Function

```
IF NOT a% THEN PRINT " a% is worth 0 "
```

Check also

Operators Priority, p. 17

3.55 NTPSYNC

Syntax

```
NTPSYNC
```

Description

This function posts a request for clock resynchronization. It occurs even if this feature is disabled in the configuration.

3.56 ONxxxxxx

There are multiple *ONxxxxxx* commands that can be used in the BASIC scripting.

These commands are used to register a BASIC action to perform in case of special conditions. For each *ONxxxxxx* command, the action to execute is a string that is used as a BASIC command line.

When the condition is met, the command is queued in an execution queue and is executed when on top of this queue.

These functions are:

ONTIMER	Executed when one of the timers expires.
ONCHANGE	Executed when a tag changes. Valid for a change of value or configuration.
ONALARM	Executed when a tag alarm state changes.
ONERROR	Executed when an error occurs during BASIC execution.
ONSTATUS	Executed when a scheduled action is ended whether the state.
ONSMS	Executed when a SMS is received (only for eWON with GSM/GPRS modem).
ONPPP	Executed when the PPP connection goes online or offline.
ONVPN	Executed when the VPN goes connected or disconnected.
ONWAN	Executed when the WAN goes connected or disconnected.
ONDATE	Executed when the defined pattern meets current the date / time.
ONMQTT	Execution of a callback when the MQTT client receives data.
ONMQTTSTATUS	Registration of a callback when the MQTT client connection status changes.

When the command line programmed is executed, a special parameter is set in *SETSYS PRG,"EVTINFO"*. The value of the parameter depends on the ONxxxxxx function and it can be checked with the *GETSYS* command.

> **!** For all *ONxxxx* command, if the last parameter is omitted, the action is canceled.

Example 90: Canceling an ONxxxxx command

```
ONTIMER 1
// This will cancel any action set on TIMER 1
```

Check also

GETSYS, SETSYS, p. 44

3.56.1 ONALARM

Syntax

```
ONALARM S1, S2
```

S1	the tag reference (tag name, ID or index).
S2	the command line to execute in case the alarm status changes.

Description

This command executes the *S2* command line when the alarm state of *S1* changes. The *EVTINFO* parameter is set to the tag ID when command is called.

ONALARM will execute the command when the alarm status gets the value “2” (or above) which means that *ONALARM* does not detect the "pre trigger" status (value = 1).

Example 91: ONALARM

```
ONALARM "MyTag", "GOTO MyTagAlarm"
```

Check also

ALSTAT, p. 23; *GETSYS, SETSYS, p. 44*; *ONCHANGE, p. 73*

3.56.2 ONCHANGE

Syntax

```
ONCHANGE S1, S2
```

S1	the tag reference (tag name, ID or index).
S2	the command line to execute in case the tag value or configuration changes.

Description

This command executes the *S2* command line when the tag *S1* changes. The change can be its value or configuration. The *EVTINFO* parameter is set to the tag ID when command is called.

ONALARM will execute the command when the alarm status gets the value “2” (or above) which means that *ONALARM* does not detect the "pre trigger" status (value = 1).

Example 92: ONCHANGE

```
ONCHANGE "MyTag", "GOTO MyTagChange"
```

Check also

IOMOD, p. 58; *GETSYS, SETSYS, p. 44*;

3.56.3 ONDATE

Syntax

```
ONDATE I1, S1, S2
```

I1	the planner entry index to set. This entry is contain between 1 to 10. If *S1* and *S2* are not provided, this planner entry is deleted instead of being set.
S1	the timer interval.
S2	the command line to execute in case the tag value or configuration changes.

Description

The *ONDATE* function allows the definition of 10 planned tasks.

All *ONDATE* entries are deleted automatically when the program is stopped by the *RUN/STOP* button.

The syntax of the *S1* parameter is the following: **mm hh dd MMM DDD**

Syntax of the S1 parameter

Field	Settings
mm	This is the “minute” parameter. A number between 0 — 59
hh	This is the “hour” parameter. A number between 0 — 23
dd	This is the “day” parameter. A number between 1 — 31
MMM	This is the “month” parameter. A number between 1 — 12 or the month name abbreviation in English (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec).
DDD	This is the “day of week” parameter. A number between 1 — 7 starting with 1 = Monday and ending with 7 = Sunday or the day name abbreviation in English (mon, tue, wed, thu, fri, sat, sun).

> **!** If *S1* is provided, all 5 parameters are required! When used together, the *dd* and *DDD* parameters operate as an OR operation: every *dd* of the *month* OR *DDD*).

In addition, there are some operators to specify multiple date/time.

*****	The * (asterisk) operator specifies all possible values for a field from the table “Syntax of the S1 parameter”. For example, an * in the *hh* time field would be equivalent to “every hour”.
,	The , (comma) operator specifies a list of values. For example: “1,3,4,7,8”. Be careful: space inside the list must not be used.
–	The - (dash) operator specifies a range of values. For example: “1-6” which is equivalent to “1,2,3,4,5,6”.
/	The / (slash) operator, called “step”, can be used to skip a given number of values. For example, “*/3” in the hour time field is equivalent to “0,3,6,9,12,15,18,21”.

Example 93: ONDATE

```
ONDATE 1,"* * * * *","GOTO MyFunc"
//  "GOTO MyFunc" every minutes.

ONDATE 1,"0 * * * *","GOTO MyFunc"
// "GOTO MyFunc" every hour.

ONDATE 1,"0 0 * * *","GOTO MyFunc"
// "GOTO MyFunc" on every day at midnight (00:00).

ONDATE 1,"*/15 * * * *","GOTO MyFunc"
// "GOTO MyFunc" every 15 minutes.

ONDATE 1,"15 7 1 1 *","GOTO MyFunc"
// "GOTO MyFunc" at 7:15, the first of january.
// Could have also be written as '15 7 1 jan *'

ONDATE 1,"15 8 * * 1","GOTO MyFunc"
// GOTO MyFunc" at 8:15, each monday.
// Could have also be written as '15 8 * * mon'

ONDATE 1,"0 8-18 * * 1-5","GOTO MyFunc"
// "GOTO MyFunc" at every hour between 8:00 and 18:00
//on every working day (Monday to Friday)

ONDATE 1,"0 6,7,18,19 * * *","GOTO MyFunc"
// "GOTO MyFunc" at 6, 7, 18 and 19 o’clock on every day.

ONDATE 1,"* * 13 * fri","GOTO MyFunc"
// "GOTO MyFunc" at every minutes on each Friday
// OR the 13th of the month (and not only on the Friday 13th).

ONDATE 1
// Will delete the planned entry 1
```

Check also

ONTIMER, p. 78; *TSET, p. 102*

3.56.4 ONERROR

Syntax

```
ONERROR S1
```

S1	the command line to execute when an error occurs during program execution.

Description

The *EVTINFO* parameter is set to the code of the error.

Example 94: ONERROR

```
ONERROR "GOTO TrapError"
```

Check also

GETSYS, SETSYS, p. 44;

3.56.5 ONMQTT

Syntax

```
ONMQTT S1
```

S1	The command line to execute when data is exchanged with the MQTT client.

Description

This command allows the execution of a callback when the MQTT client receives data related to an event, for example one or several messages received on a subscribed topic.

Example 95: ONMQTT

```
ONMQTT "GOTO ProcessMqtt" // Goto to a label
```

Check also

MQTT, p. 64

3.56.6 ONMQTTSTATUS

Syntax

```
ONMQTTSTATUS S1
```

S1	The command line to execute when the status of the MQTT client changes.

Description

This allows the registration of a callback when the MQTT client connection status changes. The connection is asynchronous and the device handles the reconnections.

Each time the MQTT connection status changes, *S1* is called.

Usually, the callback is executed:

- on disconnection, related to *MQTT "status"* = 4
- on connection, related to *MQTT "status"* = 5
- and background transparent reconnection, related to MQTT "status" = 5

If there are a lot of reconnections, this could mean that the server keepalive and client keepalive are not synced. This is not a problem, some servers will disconnect the clients if traffic isn't detected after a little time. This could happen for example if the client keepalive > server keepalive. It would be the perfect application to keep a permanent MQTT connection and reduce the heartbeat data consumption.

Example 96: ONMQTTSTATUS

```
ONMQTTSTATUS "@AWSMQTTStatusChange()" // Load a function
ONMQTTSTATUS "GOTO ProcessMqttStatus" // Goto to a label
```

Check also

MQTT, p. 64

3.56.7 ONPPP

Syntax

```
ONPPP S1
```

S1	the command line to execute when the PPP connection goes online or offline.

Description

The *EVTINFO* parameter is set to one of the following values:

1	The PPP connection has gone online.
2	The PPP has gone offline.

Example 97: ONPPP

```
ONPPP "GOTO PppAction"

PppAction:
I% = GETSYS PRG, "EVTINFO"
IF I% = 1 THEN
  PRINT "Online with address " ; GETSYS PRG, "PPPIP"
ELSE
  PRINT "PPP Going offline"
ENDIF
END
```

Check also

GETSYS, SETSYS, p. 44;

3.56.8 ONSMS

Syntax

```
ONSMS S1
```

S1	The command line to execute when the device receives an SMS.

Description

A typical use of the *ONSMS* syntax is to allow the device to send a read SMS receipt to the SMS sender.

You can read the received SMS with *GETSYS PRG* function with

smsRead	Holds 1 if there is a new SMS. Reading *smsRead* loads the other parameters. Holds 0 if the SMS queue is empty.
smsFrom	String holding the phone number of the sender.
smsDate	String holding the date of SMS reception.
smsMsg	String holding the SMS message.

Example 98: ONSMS

```
ONSMS "Goto Hsms"

Hsms:
a% = GETSYS PRG, "SmsRead"
IF (a%<>0) THEN
  s% = s%+1
  PRINT "SMS Nr: " ; s%
  f$ = GETSYS PRG, "smsfrom"
  PRINT "From: " ; f$
  PRINT GETSYS PRG, "smsdate"
  a$ = getsys prg,"smsmsg"
  PRINT "Message: " ; a$
  b$ = f$ + ",gsm,0"
  c$ = "Message received: " + a$
  sendsms b$, c$
  goto HSms
ENDIF
END
```

3.56.9 ONSTATUS

Syntax

```
ONSTATUS S1
```

S1	The command line to execute when a scheduled action is finished.

Description

The *EVTINFO* parameter is set to the *ACTIONID* of the finished action when command is called. This function can be used to track success or failure of scheduled actions.

Example 99: ONSTATUS

```
ONSTATUS "goto Status"
```

Check also

GETSYS, SETSYS, p. 44; *PUTFTP, p. 90*; *SENDMAIL, p. 93*; *SENDSMS, p. 94*; *SENDTRAP, p. 95*

3.56.10 ONTIMER

Syntax

```
ONTIMER E1, S1
```

E1	The timer number initiated by *TSET* command.
S1	The command line to execute when timer expires.

Description

This command executes *S1* command line when *E1* expires.

The *EVTINFO* parameter is set to the timer number when command is called.

Example 100: ONTIMER

```
ONTIMER 1, "goto Timer1"
ONTIMER 1, "LOGIO 'mytag'"
```

Check also

GETSYS, SETSYS, p. 44; *TSET, p. 102*

3.56.11 ONVPN

Syntax

```
ONVPN S1
```

S1	The command line to execute when the VPN connection status change: connection or disconnection.

Description

The *EVTINFO* parameter is set to one of the following values:

1	The VPN connection has gone online.
2	The VPN connection has gone offline.

Example 101: ONVPN

```
ONVPN "goto VPN_Action"

VPN_Action:
i% = GETSYS PRG, "EVTINFO"
IF I%=1 THEN
  PRINT "VPN Online"
ELSE
  PRINT "VPN Going offline"
ENDIF
END
```

Check also

GETSYS, SETSYS, p. 44;

3.56.12 ONWAN

Syntax

```
ONWAN S1
```

S1	The command line to execute when the WAN connection status change: connection or disconnection.

Description

The *EVTINFO* parameter is set to one of the following values:

1	The WAN connection is connected.
2	The WAN connection is disconnected.

If the WAN connection is established through an Ethernet cable, the WAN connection is not considered as disconnected if this Ethernet cable is plugged out.

Example 102: ONWAN

```
ONWAN "goto WAN_Action"

WAN_Action:
i% = GETSYS PRG, "EVTINFO"
IF I%=1 THEN
  PRINT "WAN online with address" ; GETSYS PRG, "WANIP"
ELSE
  PRINT "WAN going offline"
ENDIF
END
```

Check also

GETSYS, SETSYS, p. 44;

3.57 OPEN

Files accessed in BASIC can be of 4 different types:

- Files from the /usr directory
- Serial communication link
- TCP or UDP socket
- Export Block Descriptor

There are two different modes of operation for the file access:

- Binary mode: file is read by blocks of bytes.
- Text mode: files are read or written as CSV files.

See the *GET, p. 39* and *PUT, p. 88* commands for a detailed difference between the BINARY and TEXT mode outputs.

There are 3 operation types:

Operation types	
Parameter Value	**Description**
INPUT	The file must exist. It is opened for a read only operation. The file pointer is set to the beginning of the file.
OUTPUT	The path must exist. If the file exists, it is erased first. The file is opened for write only operation.
APPEND	The path must exist. The file doesn't have to exist. If the file does not exist, it is created (like the OUTPUT type). If the file exists, it is opened and the write pointer is located at the end of the file. The file is opened for write only operation.

When binary mode is used, the data written to the file are strings of characters that are considered as stream of bytes.

The *GET* command returns the amount of bytes requested. When text mode is used, the operation is completely different: the *PUT* operation is more like a *PRINT* command directed to the file, the data is formatted as text and each data is separated by a “;” in the output file (strings are exported between quotes).

The *GET* command works like a READ command: the file is read sequentially and each *GET* returns one of the “;” separated element. The type of the data returned depends on the type of data read.

In both modes, files are read sequentially until the end of file is reached. The end of file can be tested with the *EOF* function.

The device user flash file system allows up to 8 files to be simultaneously opened for read (even twice the same file), and 1 file opened for write. If a file is opened for read, it cannot be opened for write at the same time (and vice versa).

Running the program will close all files previously opened (not the *GOTO*).

Check also

CLOSE, p. 26; *EOF, p. 29*; *GET, p. 39*; *PUT, p. 88*

3.57.1 File OPEN /usr

Syntax

```
OPEN S1 FOR BINARY|TEXT INPUT|OUTPUT|APPEND AS E1
```

S1	describes the access to a file that is located on the device directories.
E1	the file number.

Description

After the *OPEN* operation, the file is referenced by *E1* and no longer by its file name. There are 8 file numbers available. Once a file number is assigned to a file, it is allocated to that specific file until the *CLOSE* command is issued.

S1 must respect the following syntax:

```
"file:/directory/filename"
```

This allows to read or write files in the /usr directories. The root, containing virtual files such as config.txt, comcfg.txt... can not be accessed through this command.

Example 103: OPEN a File in /usr folder

```
OPEN "file:/sys/test.dat" FOR BINARY INPUT AS 1
a$ = GET 1, 4
CLOSE 1
// This opens file 1 Reads 4 bytes
```

3.57.2 TCP or UDP Stream OPEN

Syntax

```
OPEN S1 FOR BINARY INPUT|OUTPUT AS E1
```

S1	describes the access to a stream.
E1	the file number.

Description

This command works only with BINARY.

After the *OPEN* operation, the file is referenced by *E1* and no longer by its file name. There are 8 file numbers available. Once a file number is assigned to a file, it is allocated to that specific file until the *CLOSE* command is issued.

S1 must respect the following syntax:

```
"tcp:Address:destPort[, TimeOut]"
"udp:Address:dest_Port[:srcPort][, TimeOut]"
```

Address	a dotted IP address like 10.0.0.1 or a valid resolvable internet name such as www.ewon.biz.
destPort	a valid port number from 1 to 65535.
srcPort	if defined, the return port will be forced to the srcPort value (works only with UDP protocol). if not defined, the return port is allocated automatically by the device TCP/IP stack.
TimeOut	the number of seconds the device will wait to decide if the *OPEN* command failed (default : 75 sec).

For scheduled action, when the *OPEN* command is used to initiate a TCP connection, the command returns before the connection is actually opened.

A scheduled action is posted because opening the socket may require a dial out or take more than a minute as the BASIC cannot be stopped during that time.

To check if the connection is established, 2 options are possible:

- Verify the scheduled action status by checking the *PRG, ACTIONSTAT* (See *GETSYS, SETSYS, p. 44*).
- Read the file with *GET*: as long as the file is not actually opened, the function returns *#CLOSED#*. When the function stops sending *#CLOSED#* the file can be read and written for socket operations.

Example 104: OPEN a Stream

```
OPEN "tcp:10.0.0.1:25" FOR BINARY OUTPUT AS 1
PUT 1, CHR$(13) + CHR$(10)
a$ = GET 1
CLOSE 1
// Opens socket to 10.0.0.1 port 25 for read/write access.
// Writes a CRLF then reads response.
```

3.57.3 COM Port OPEN

Syntax

```
OPEN S1 FOR BINARY INPUT|OUTPUT AS E1
```

S1	describes the access to a stream.
E1	the file number.

Description

This command works only with *BINARY*. Both *INPUT* and *OUTPUT* modes allow to both read and write on the COM port.

Attempting to use a serial port already taken by an IO server is not allowed and returns an error.

This command will open the serial port from 1 to 4 with the given line parameters.

After the *OPEN* operation, the file is referenced by *E1* and no longer by its file name. There are 8 file numbers available. Once a file number is assigned to a file, it is allocated to that specific file until the *CLOSE* command is issued.

S1 must respect the following syntax:

```
com:n,b,dpsh
```

n	number between 1 to 4. The port number 1 is the front panel serial port, the 2 is the modem port.
b	the baud rate
d	the number of bits “7” or “8”
p	the parity: “e”, “o” or “n”
s	the number of stop bit “1” or “2”
h	the handshaking where “h” is half duplex, “r” is yes Rts/Cts and “n” is No

Example 105: OPEN file on a COM port

```
OPEN "com:1, 9600, 8n1n" FOR BINARY OUTPUT AS 3
// Opens the Serial port 1 with speed 9600,
// bit 8, parity none, stop bit 1 and handshaking no.
```

3.57.4 Export Block Descriptor OPEN

Syntax

```
OPEN S1 FOR TEXT|BINARY INPUT AS E1
```

S1	the export block descriptor. Must be under the form "exp:XXXXX", where XXXXX is an EBD.
E1	the file number.

Description

After the *OPEN* operation, the file is referenced by *E1* and no longer by its file name. There are 8 file numbers available. Once a file number is assigned to a file, it is allocated to that specific file until the *CLOSE* command is issued.

When the EBD has been read (or not if the command is closed before the end), the *CLOSE* command must be called to release memory.

The PUT command can not be used with this type of OPEN.

Example 106: OPEN with Export Block Descriptor

```
OPEN "exp:$dtAR $ftT" FOR TEXT INPUT AS 1

Loop:
a$ = Get 1
PRINT a$
IF a$ <> "" THEN GOTO Loop
CLOSE 1
```

In the example above, *a$ = GET 1* can be called until it returns an empty string to read the content of the Export Block Descriptor. The data is then read by blocks of maximum 2048 bytes.

If the size needs to be reduced or increased, the call should be *a$ = GET 1, y*, where *y* is the maximum number of bytes the function should return. If *y* is 0, it should be omitted.

Example 107: OPEN with Export Block Descriptor

```
OPEN "exp:$dtUF $ftT $fn/myfile.txt" FOR TEXT INPUT AS 1
a$ = GET 1
PRINT a$
CLOSE 1
```

3.58 OR

Syntax

```
E1 OR E2
```

Description

This operator does a bit-by-bit *OR* between the 2 integers *E1* and *E2*.

The behavior depends on the nature of *E1* and *E2*:

- When executed on float elements (float constant or float variable), the *OR* function returns the logical *OR* operation.
- When executed on integer elements (integer constant or integer variable - like *i%*), the *OR* function returns the bitwise *OR* operation

This behavior doesn't apply on *AND* and *XOR*.

Example 108: OR Operator

```
1 OR 2 // returns 3
2 OR 2 // returns 2
3 OR 1 // returns 3

// Logical OR
a = 0.0
b = 0.0
ORResult = a OR b
PRINT ORResult  // Prints 0.0

c = 0.0
d = 12.0
ORResult = c OR d
PRINT ORResult  // Prints 1.0
```

Check also

Operators Priority, p. 17; *AND, p. 24*; *XOR, p. 105*

3.59 PI

Syntax

```
PI
```

Description

This function returns the PI number: 3.14159265

3.60 PRINT – AT

Syntax

```
PRINT [AT] [E1, E2] CA[1;[CA2]]
```

The PRINT command accepts multiple syntaxes:

PRINT CA	displays *CA* followed by a new line
PRINT CA;	displays *CA* without a new line
PRINT AT E1, E2 CA	displays *CA* at the *E1* column and at the *E2* line
PRINT CA1;CA2;CA3...	display the *CA1*, *CA2*... one following the other without going to the next line.

Description

The device has a “virtual screen” that can be used to inspect the content of values while the program is running or to debug an expression.

The PRINT command cannot be followed immediately by parenthesis ().

Example 109: PRINT AT

```
PRINT "HOP1" ; "HOP2"
```

Check also

CLS, p. 26

3.61 PRINT

Syntax

```
PRINT #x, CA
```

#x	where x can have 2 values: • *0*: targets the user’s web page • *1*: targets the virtual screen
CA	the variable, text, ... to print

Description

The *PRINT* command sends output to the virtual screen.

With the *PRINT #* command, output can be routed to another destination. When running ASP code, the *PRINT* command can be used to build the content of the page sent to the user.

If the *PRINT* is sent to a web page, an HTML tag “
” is automatically added at the end of line to pass to the next line. If the return cage shouldn’t be added, a “;” (semicolon) must be added after the *CA*.

The *PRINT* command cannot be followed immediately by parenthesis ().

Example 110: PRINT with target

```
PRINT #0, a$  // Sends a$ to the user's web page
PRINT #1, a$  // Works like PRINT a$ by sending to the virtual screen
```

3.62 PUT

The put command works completely differently if the file is opened in binary mode or in text mode. The file must be opened for *OUTPUT* or for *APPEND* operation (*APPEND* is for /usr files only).

The command description describes operation for /usr (text and binary modes), COM (always binary) and TCP-UDP (always binary).

Check also

CLOSE, p. 26; *EOF, p. 29*; *GET, p. 39*; *OPEN, p. 80*

3.62.1 File – Binar Mode

Syntax

```
PUT E1, S1[; S2...]
```

E1	*E1* is the file number: 1 — 8
S1	the string of char. to append to the file. The number of bytes written depends on the length of the string.
S2	additional data to write. The length of a BASIC line limits the number of items.

The delimiter between the file number and the first item is a “,” (comma) but the separator between the first item and the optional next item is a “;” (semicolon). This is close to the *PRINT* syntax.

Example 111: PUT for a File in Binary Mode

```
OPEN "/myfile.bin" FOR BINARY OUTPUT AS 1
PUT 1, "ABCDEF"; "GHIJKLMN"
CLOSE 1
// Now reopen and append
OPEN "/myfile.bin" FOR BINARY APPEND AS 1
PUT 1, "OPQRSTUVWXYZ"
CLOSE 1
```

3.62.2 File – Text Mode

Syntax

```
PUT E1, V1[; V2…][;]
```

E1	*E1* is the file number: 1 — 8
V1	an element of type string, integer or float
S2	additional data to write (string, integer or float)

Description

The data is converted to text before being written to file. If data is of string type it is written between quotes, otherwise not. A semicolon is inserted between each data written to the file.

If the *PUT* command line ends with a semicolon, the sequence of data can continue on another BASIC line. If the *PUT* command line ends without the semicolon character, the line is considered as finished and a *CRLF (CHR$(13)+CHR$(10))* is added to the end of the file.

Example 112: PUT for a File in Text Mode

```
OPEN "/myfile.txt" FOR TEXT OUTPUT AS 1
PUT 1, 123; "ABC";
PUT 1, "DEF"
PUT 1, 345.7; "YYY"; "ZZZ"
CLOSE 1

// This would produce the file:
// 123;"ABC";"DEF"
// 345.7;"YYY";"ZZZ"
```

3.62.3 COM – Binary Mode

Syntax

```
PUT 1, S1
```

S1	string of data to write to serial port

Description

This command writes the *S1* string to the serial port. The function returns only after all the data has been sent.

The string can contain any byte by using the *CHR$* function. Serial port cannot be used by an IO server at the same time, or it would result in an “IO Error”.

Example 113: PUT to a COM port in Binary Mode

```
OPEN "/myfile.txt" FOR TEXT OUTPUT AS 1
PUT 1, 123; "ABC";
PUT 1, "DEF"
PUT 1, 345.7; "YYY"; "ZZZ"
CLOSE 1
// This would produce the file:
// 123;"ABC";"DEF"
// 345.7;"YYY";"ZZZ"
```

3.62.4 TCP/UDP – Binary Mode

Syntax

```
PUT E1, S1
```

E1	the file number returned by the *OPEN* function
S1	the string of data to write to the socket

Description

This command writes *S1* to the socket. The string can contain any byte by using the *CHR$* function.

The function returns only after all the data has been actually transferred to the stack.

The socket must be opened. The *OPEN* command returns immediately but generates a scheduled action. The *PUT* command will generate an IO error until the socket is actually opened.

When data is transferred to the TCP/IP stack, it does not mean that the data has been received by the socket end point. It may take a little time for the data to be considered as undeliverable and the socket to bet set in error mode.

3.63 PUTFTP

Syntax

```
PUTFTP S1, S2[, S3]
```

S1	The destination file name, to write on the FTP server.
S2	The file content of string type. This content may also be an Export Block Descriptor content.
S3	The FTP server connection parameters. If unused, the FTP server parameters from the “General Configuration” of the web interface of the device will be used.

Description

This command puts a file on an FTP server. The content of the file is either a string or an Export Bloc Descriptor.

The *S3* parameters is as follow

```
[user:password@]servername[:port][,option1]
```

option1	This option sets the mode of transmission: value 1 is passive mode where value 2 is active mode. If omitted, option1 = 0, and the mode will be set to active mode.

This command posts a scheduled action request for a *PUTFTP* generation.

When the function returns, the *GETSYS PRG, "ACTIONID"* returns the ID of the scheduled action and allows the tracking of this action.

It is also possible to program an *ONSTATUS* action that will be called when the action is finished (with or without success).

Example 114: Put a File on a FTP Server

```
// Post a file containing a custom text
a$ = "/ewon1/MyFile.txt"
b$ = "this is the text of the file"
PUTFTP a$, b$

// Post a file containing the event log
a$ = "/ewon1/events.txt"
b$ = "[$dtEV]"
PUTFTP a$, b$

// Post a file with the Histo logging of Temperature tag
// on a defined FTP server.
a$ = "/ewon1/Temperature.txt"
b$ = "[dtHL$ftT$tnTemperature]"
c$ = "user:pwd@FTPserver.com"
PUTFTP a$, b$, c$
```

Check also

GETSYS, SETSYS, p. 44; *ONxxxxxx, p. 72*; *ONSTATUS, p. 78*

3.64 PUTHTTP

Syntax

```
PUTHTTP S1, S2, S3, S4, S5[, S6]
```

S1	The connexion parameter with the format: [user:password@]servername[:port]
S2	The URI of the action (absolute path of the request URI).
S3	The text fields with the format: fieldname1=valuename1[&fieldnameX=valuenameX&...]
S4	The file fields with the format: fieldname1=exportblockdescriptor1[&fFieldnameX= exportblockdescriptorX&...]
S5	The error string.
S6	The proxy information

Description

The *PUTHTTP* command submits an HTTP form to a web server (just as if a web form was sent on a website). The submitted forms may contain text fields and file fields.

The HTTP method used is the POST method (multipart/form-data). Content Type of the file fields is always application/octet-stream. Files to upload are defined using the Export Block Descriptor syntax.

When this function returns, the *GETSYS PRG* function returns the ID of the scheduled action and allows the tracking of this action. It is also possible to program an *ONSTATUS* action that will be called when the action is finished (with or without success).

When "*PROXY*" is added at the end of the command, the device performs the PUTHTTP through a proxy server. The device will use the proxy server parameters configured in System Setup / Communication / VPN Global.

There are some rules to follow in the syntax:

- All the parameters are mandatory. If a text field is not needed, the *S3* parameter should be transmitted as an empty string.
- When file fields aren't needed, an empty string is used for *S4*. When no port is specified HTTP port 80 is used.
- The HTTP server response will be checked against the *S5*. If the response contains *S5*, the command will finish without success.
- Spaces in text fields and file fields are not allowed except inside export block descriptors (inside the EBD brackets).
- One " fieldname=valuename" section in the text field parameter may not exceed 7500 bytes, otherwise action will finish without success. This limitation does not apply for the file fields.

> **!** The posting method used (chunked packets) is only correctly handled on IIS 6.0 and Apache web servers. Posting on IIS 5 doesn't work (i.e: Windows XP). Chuncked packets are not applied when the *"PROXY"* parameter is used because most proxy servers do not accept them.
>
> If *PUTHTTP* is used with the *"PROXY"* parameter, then device creates a temporary file named "puthttp.proxy" in the /usr directory to store the data locally before sending it towards the server via the proxy.

Example 115: PUTHTTP

```
b$ = "/textfields.php"
c$ = "firstname=james&lastname=smith"
e$ = "failed"

// Text fields form without HTTP basic authentication
a$ = "10.0.5.33"
d$ = ""
PUTHTTP a$, b$, c$, d$, e$

// Text fields with basic authentication and dedicated HTTP port
a$ = "adm1:adm2@www.ewon.biz:89"
d$ = ""
PUTHTTP a$, b$, c$, d$, e$

// Text fields + file fields
a$ = "10.0.5.33"
d$ = "pictures[]=[$dtEV $fnevents.txt]&pictures[]=[...]"
PUTHTTP a$, "/upload.php", c$, d$, e$

// Text fields without HTTP basic authentication through proxy
a$ = "10.0.5.33"
d$ = ""
f$ = "PROXY"
PUTHTTP a$, b$, c$, d$, e$, f$
```

Check also

GETHTTP, p. 43; *GETSYS, SETSYS, p. 44*; *ONSTATUS, p. 78*

3.65 REBOOT

Syntax

```
REBOOT
```

Description

This Basic keyword provides a very easy way to reboot the device.

A typical use of this command is by writing it into a file named "remote.bas", saving it locally and uploading this file on the FTP of the device to replace the existing "remote.bas" file. The device will reboot directly.

3.66 REM

Syntax

```
REM free text
```

Description

This command enables the insertion of a line of comment in the program. The interpreter does not consider the line.

Example 116: Insert a Remark

```
REM This line will not be considered
a% = 2  REM Neither will this second part of the line
```

Check also

// (comment), p. 22

3.67 RENAME

Syntax

```
RENAME S1, S2
```

Description

This command changes the name of file *S1* to *S2*. The command only works in the /usr directory. Omitting “/usr/” before the filename will result in an IO error.

The file and directory names are case sensitive. The directory must exist before the call of the function as there is no automatic directory creation.

Example 117: Rename a File

```
RENAME "/usr/OldName.txt", "/usr/NewName.txt"
```

Check also

ERASE, p. 30

3.68 RTRIM

Syntax

```
RTRIM S1
```

Description

This command returns a copy of a string with the rightmost spaces removed.

Example 118: Trim on the Right

```
b$ = RTRIM a$
```

Check also

LTRIM, p. 63

3.69 SENDMAIL

Syntax

```
SENDMAIL S1, S2, S3, S4
```

S1	The email address of the recipients (TO). Multiple recipients can be set, separated by a semicolon.
S2	The email address of the recipient Carbon Copies (CC). Multiple recipients can be set, separated by a semicolon.
S3	The subject of the message.
S4	The content of the message.

Description

This command posts a scheduled action request for an email generation. When the function returns, the *GETSYS PRG, "ACTIONID"* returns the ID of the scheduled action and allows the tracking of this action.

It is also possible to program an *ONSTATUS* action that will be called when the action is finished (with or without success).

The *S4* content follows a special syntax that allows the insertion of an Export Block Descriptor inside the content itself or as attachment. This syntax is:

- [EXPORT_BLOCK_DESCRIPTOR]: will be replaced by the corresponding data and put inside the message.
- &[EXPORT_BLOCK_DESCRIPTOR]: will be set as attachment.

Example 119: Send a Mail

```
m$ = "Event Log data are attached to this mail &[$dtEV]"
// Email content: "Event Log data are attached to this mail"
// Email attachment: events log

SENDMAIL "ewon@actl.be", "", "Subject", "Message"
SENDMAIL "ewon@actl.be", "", "Subject", m$
```

Check also

GETSYS, SETSYS, p. 44; *ONxxxxxx, p. 72*; *ONSTATUS, p. 78*

3.70 SENDSMS

Syntax

```
SENDSMS S1, S2
```

S1	The SMS recipients list.
S2	The content of the message (maximum 140 characters).

Description

This command posts a scheduled action request for an SMS generation.

When the function returns, the *GETSYS PRG, "ACTIONID"* returns the ID of the scheduled action and allows the tracking of this action. It is also possible to program an *ONSTATUS* action that will be called when the action is finished (with or without success).

For the syntax of S1, refer to the chapter "SMS on alarm configuration" in the General Reference Guide.

Example 120: Send an SMS

```
// Send an SMS to 2 recipients.
d$ = "0407886633,ucp,0475161622,proximus"
d$ = d$ + ";" + "0407886634,gsm,0"
SENDSMS d$, "Message from eWON"
```

Check also

GETSYS, SETSYS, p. 44; *ONxxxxxx, p. 72*; *ONSTATUS, p. 78*

3.71 SENDTRAP

Syntax

```
SENDTRAP E1, S1
```

E1	The first trap parameter.
S1	The second trap parameter.

Description

This command posts a scheduled action request for an SNMP trap generation.

The first parameter is sent on OID .1.3.6.1.4.1.8284.2.1.4.2

The second parameter is sent on OID .1.3.6.1.4.1.8284.2.1.4.1

```
-
-- Script information
-- ewonScript OBJECT IDENTIFIER ::= { prodEwon 4 }

scpUserNotif OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..255))
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"This is the text of the last trap sent by the Script"
::= { ewonScript 1 }

scpUserNotifI
OBJECT-TYPE SYNTAX Integer32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"This is a free parameters for script generated traps"
::= { ewonScript 2 }
```

When the function returns, the *GETSYS PRG, "ACTIONID"* returns the ID of the scheduled action and allows the tracking of this action. It is also possible to program an *ONSTATUS* action that will be called when the action is finished (with or without success).

Example 121: Send an SNMP Trap

```
// Send a trap with NotifI = 10 and Notif = Trap message
SENDTRAP 10, "Trap message"
```

Check also

GETSYS, SETSYS, p. 44; *ONxxxxxx, p. 72*; *ONSTATUS, p. 78*

3.72 SETIO

Syntax

```
SETIO S1, F1
```

S1	The tag reference (tag name, ID or index).
F1	The value that will be set to *S1*.

Description

This command modifies the value of a tag. The tag must be writable (not for the read-only Tags).

In many cases, this function is efficiently replaced by the TagName@ syntax.For example SETIO "MyTag", 10.2 is equivalent to MyTag@=10.2

Example 122: Set a Tag Value

```
SETIO "myTag", 10.123
```

3.73 SETTIME

Syntax

```
SETTIME S1
```

S1	The new date / time to set.

Description

Updates the real time clock of the device.

S1 can contain only the time. In that case, the date is not modified. It can also contain only a date. In that case, the time is set to 00:00:00

An event is generated in the events log when using this command.

Example 123: Set the Date and / or Time

```
SETTIME "13/12/2017" // Time is set to 13/12/2017 00:00:00
SETTIME "13/12/2017 12:00" // Time is set to 13/12/2017 12:00:00
PRINT TIME$ // Returns for example "15/01/2000 07:38:04"
SETTIME "12:00" // Time is set to 15/01/2017 12:00:00
```

Check also

TIME$, p. 101

3.74 SFMT

Syntax

```
FCNV E1|F1, EType[, ESize, SFormat]
```

E1|F1	The integer or float to format into string.
EType	The parameter determining the type of conversion.
ESize	The size of the string to convert.
SFormat	The format specifier for the conversion

Description

Converts a number (float or integer) to a formatted string. The type of conversion is determined by the *EType* parameter.

If *ESize* is equal to 0 (or negative) with an *SFormat* present, then *ESize* is the size of the output string as formatted.

If *ESize* is positive, *SFMT* will produce a string of *ESize* bytes.

Etype value	Conversion type
1	convert float number to string holding the IEEE representation (MSB first)
2	convert float number to string holding the IEEE representation (LSB first)
10	convert integer to string (MSB first)
11	convert integer to string (LSB first)
20	format float number using an *SFormat* specifier
30	format integer number using an *SFormat* specifier
40	format time as integer into time as string

Each of the *EType* is explained and described in the following sub-chapters.

Check also

FCNV, p. 31

3.74.1 Convert Float to an IEEE Representation

The IEEE float representation use four bytes (32 bits).

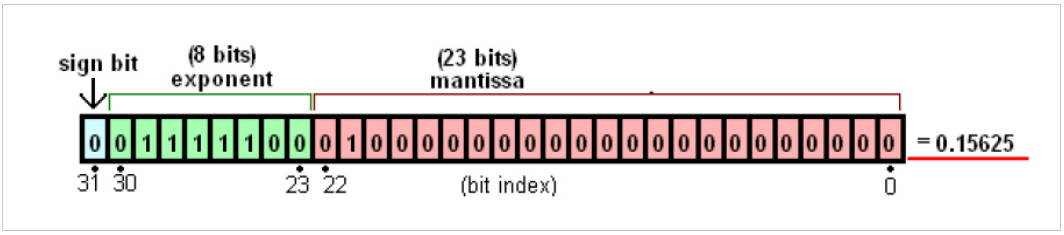

Fig. 2 Conversion to an IEEE Float

EType is equal to 1 or 2.

The string could be LSB (Least Significant Byte) first which will convert *FloatNum* to a string holding the IEEE representation with MSB (Most Significant Byte) first.

```
a$ = SFMT FloatNum, 1
a$(1)  // MSB which represents Exponent + Sign
```

```
...
a$(4)  // LSB which represents Mantissa
```

The string could also be MSB first which will convert *FloatNum* to a string holding the IEEE representation with LSB first

```
a$ = SFMT FloatNum, 2
a$(1)  // LSB which represents Mantissa
...
a$(4)  // MSB which represents Exponent + Sign
```

Example 124: Conversion from an IEEE Float Variable

```
ieee = -63.456
a$ = SFMT ieee, 1
// a$(1) = 194; a$(2) = 125; a$(3) = 210, a$(4) = 242

a$ = SFMT ieee, 2
// a$(1) = 242; a$(2) = 210; a$(3) = 125; a$(4) = 194
```

3.74.2 Compute an Integer to a String

Convert an integer value to a string holding the bytes array representation of this integer.

This representation can be MSB (Most Significant Byte) first or LSB (Least Significant Byte) first.

The *ESize* parameter is required, it is the size of the returned string (it can be 1, 2, 3 or 4).

Example 125: Compute an Integer to a String

```
a% = 1534
a$ = SFMT a%, 10, 4
// a$(1)=0; a$(2)=0; a$(3)=5; a$(4)=254

a$ = SFMT a%, 11, 4
// a$(1)=254; a$(2)=5; a$(3)=0; a$(4)=0
```

3.74.3 Convert a Float to a String using an SFormat Specifier

Convert a float number to a string using a format specifier.

The LRC computation is the sum of all bytes modulo 256.

The *ESize* parameter is required. It is the size of the returned string. Use 0 to let the device set the length.

The *SFormat* parameter is required. It is the format specifier string and is like *"%f"* or *"%.5g"*.

The syntax for the float format specifier is as follow

```
%[flags][width][.precision]type
```

type	“f” or “F”: prints a float in normal (fixed-point) notation. “e” or “E”: prints a float in standard form ([-]d.ddd e[+/-]ddd). “g” or “G”: prints a float in either normal or exponential notation (lowercase or uppercase), whichever is more appropriate for its magnitude. This type differs slightly from fixed-point notation as insignificant zeroes on the right side of the decimal separator are not included. Also, the decimal point is not included on whole numbers.
flags	“+”: always denote the sign “+” or “-” of a number (the default is to omit the sign for positive numbers). “0”: use 0 to left pad the number.
width	“number”: set the length of the whole string for padding. Only needed when flag “0” is used.
.precision	“number”: the decimal portion precision of the output that should be expressed in number digits.

Example 126: Convert a Float to a String using an SFormat Specifier

```
MyVal = 164.25
a$ = SFMT MyVal, 20, 0, "%f"  // a$ = "164.250000"
a$ = SFMT MyVal, 20, 0, "%012.3f"  // a$ = "00000164.250"
a$ = SFMT MyVal, 20, 0, "%e"  // a$ = "1.642500e+02"
```

3.74.4 Convert an Integer to a String using an SFormat Specifier

Convert an integer number to a string using a format specifier.

The *ESize* parameter is required. It is the size of the returned string. Use 0 to let the device set the length.

The *SFormat* parameter is required. It is the format specifier string and is like *"%d"* or *"%o"*.

```
%[flags][width]type
```

type	“d”: convert into integer notation. “o”: convert into Octal notation. “x” or “X”: convert into Hexadecimal notation (lowercase or uppercase).
flags	“+”: always denote the sign “+” or “-” of a number (the default is to omit the sign for positive numbers). “0”: use 0 to left pad the number.
width	“number”: set the length of the whole string for padding. Only needed when flag “0” is used.

Example 127: Convert an Integer to a String using an SFormat Specifier

```
a% = 2568
a$ = SFMT a%, 30, 0, "%010d"  // a$ = "0000002568"
a$ = SFMT a%, 30, 0, "%o"  // a$ = "5010" OCTAL notation
a$ = SFMT a%, 30, 0, "%X"  //a$ = "A08"
```

3.74.5 Convert Time as Integer into Time as String

Convert an integer holding the number of seconds since “01/01/1970 00:00:00” into a string holding a time in the format “dd/mm/yyyy hh:mm:ss”.

If the provided time is not an integer, the function will return a syntax error. If a float parameter is passed, it must be converted to an integer value first. Float value is not accurate enough to hold the big numbers used to represent seconds since “1/1/1970”, this leads to a lost of precision during time conversion.

Example 128: Convert Time as Integer into Time as String

```
a$ = SFMT 0, 40  // a$ = "01/01/1970 00:00:00"

a% = 1000000000
a$ = SFMT a%, 40  // a$ = "09/09/2001 01:46:40"
```

3.75 SGN

Syntax

```
SGN F1
```

Description

This function returns the sign of the provided float:

- If *F1* is > 0, the function returns 1.
- If *F1* = 0, the function returns 0.
- If *F1* is < 0, the function returns -1.

Example 129: Get the Sign of a Float

```
SGN (-10)  // Returns -1
SGN (-10.6)  // Returns -1
SGN 10  // Returns 1
```

3.76 SQRT

Syntax

```
SQRT F1
```

Description

This function returns the square root of *F1*. If an integer is supplied, the returned value will be a float.

Example 130: Get the Square Root of a Tag

```
SQRT 16  // Returns 4
```

3.77 STR$

Syntax

```
STR$ F1|E1
```

Description

The function returns the character string related to an *E1* or *F1* number.

Example 131: Get the Character String

```
a% = 48
a$ = STR$ a%  // a$ is worth "48"
```

Check also

VAL, p. 103

3.78 TIME$

Syntax

```
TIME$
```

Description

This command returns the string with the current date and time. The output format is “dd/mm/yyyy hh:mm:ss”.

The number of characters in the returned string is constant (19).

The *GETSYS* command provides a method to return the current time as a number of seconds since 1/1/1970. The *SFMT* and *FCNV* functions allow the conversation between time (string) and time (integer).

Example 132: Get the Current Time

```
PRINT TIME$  // Print for example 13/12/2017 11:38:58
a$ = TIME$
```

Check also

FCNV, p. 31; *SETTIME, p. 96*; *SFMT, p. 97*

3.79 TGET

Syntax

```
TGET E1
```

E1	The number of the timer (1 to 4).

Description

This command returns N (>0) if the timer expires and then resets the value. N is the number of times the timer has expired.

It returns “0” if the timer did not expired since the last call to *TGET*.

Example 133: Read a Timer with TGET

```
// Timer 1 minute
TSET 1, 60

Label1:
IF NOT TGET 1 THEN GOTO LABEL1
```

Check also

ONTIMER, p. 78; *TSET, p. 102*

3.80 TSET

Syntax

```
TSET E1, E2
```

E1	The number of the timer (1 to 4).
E2	The value in seconds of the timer.

Description

This function initializes the timer *E1* at an *E2* time base (in seconds). The timer is read by TGET.

To stop a timer, *E2* must be set to 0.

Example 134: Set a Timer associated with an Action

```
// Timer 1 minute
TSET 1, 60

Label1:
IF NOT TGET 1 THEN GOTO LABEL1
```

Check also

ONTIMER, p. 78; *TGET, p. 101*

3.81 TYPE$

Syntax

```
TYPE$(Tag|Var)
```

Tag|Var	The name of the tag or variable.

Description

This command returns the nature of the tag or the variable. Those values can be: “string”, “float” or “integer”.

As the variables are already typed, it makes more sense to use this command with tags.

3.82 VAL

Syntax

```
VAL S1
```

Description

The function evaluates the character string and returns the corresponding expression.

VAL is a function that usually takes an expression and returns a real after the expression is evaluated. It can also evaluate an expression that returns a string.

Example 135: Evaluate a Character String

```
a$ = "12"
a% = VAL("10" + a$)  // a% equals 1012
a$ = "abc"
b$ = "efg"
c$ = val("a$ + b$")  // c$ equals "abcefg"
```

Check also

STR$, p. 101

3.83 WAIT

Syntax

```
WAIT N1, S1[, N2]
```

N1	The File number to wait on.
S1	The operation to execute (max 255 char).
S2	The timeout in seconds. If omitted, the default is 60 seconds.

Description

The *WAIT* command is used to monitor events on files. The instruction would be: wait for data available on *N1* (or timeout) then execute the *S1* operation.

The monitored events are the data received on TCP and UDP socket

The *WAIT* function registers a request to wait for the event, it will not block until the event occurs.

When the *WAIT* function calls the operation, it will preset the *EVTINFO*, with the result of the operation:

EVTINFO	Signification
> 0	The event occurred and read can follow: • =1: Read is pending • =2: Ready for Write • =3: Ready for Write and Read is pending If Read is pending, the *a$ = GET N1* function will be used. In case the *GET* function returns an empty string, it means that there is an error on the socket (either the socket was closed by the other party or the socket is not writable). Following this error, the file should be closed because it is not more valid.
-1	The wait operation was aborted because of an error on the file monitored (for example the file was closed).
-2	The condition was not met during the wait operation (timeout).

A maximum of 4 *WAIT* commands can be occurring at the same time.

If a *WAIT* command is pending on a file and another *WAIT* command is issued on the same file, an “IO Error” error will occur.

Example 136: Monitor Events using WAIT

```
// This example concerns TCP socket
// and connects to a server running the ECHO protocol
Tw:
CLS
CLOSE 1
OPEN "tcp:10.0.100.1:7" FOR BINARY OUTPUT AS 1
o% = 0

wo:
a% = GETSYS PRG, "actionstat"
IF (a%=-1) THEN GOTO wo
  PUT 1, "msg_start"
  WAIT 1, "GOTO rx_data"
END

rx_data:
a% = GETSYS PRG, "evtinfo"
IF (a%>0) THEN
PRINT "info:" ; a%
a$ = GET 1
PRINT a$
PUT 1, "abc" + Str$(o%)
o% = o% + 1
WAIT 1, "GOTO rx_data"
ELSE
  PRINT "error:" ; a%
ENDIF
```

3.84 WOY

Syntax

```
WOY E1|S1
```

E1|S1 The date in integer format (number of seconds since 1/1/1970) or string format ("18/09/2003 15:45:30").

Description

This function returns an integer corresponding to the “ISO8601 Week-Of-Year” number that matches a specified time variable.

The function shouldn’t be called with a float variable as this would result in an error “invalid parameter”.

Example 137: Get the Week of Day

```
a$ = TIME$
a% = WOY a$

b% = GETSYS PRG,"TIMESEC"
a% = WOY b%
```

Check also

DAY, p. 26; *DOW, p. 28*; *DOY, p. 28*; *MONTH, p. 64*

3.85 WRITEEBD

Syntax

```
WRITEEBD S1, S2
```

S1	An Export Block Descriptor (EBD) in a string format.
S2	The file path the EBD content will be streamed in.

Description

This command streams an Export Block Descriptor (EBD) to the filesystem using a scheduled action. It returns an action ID.

The syntax of an EBD is explained in the General Reference Guide corresponding to the device.

Example 138: Use an EBD

```
WRITEEBD "$dtEV", "/usr/myEvent.txt"
```

3.86 XOR

Syntax

```
E1 XOR E2
```

S1	the tag reference (tag name, ID or index)

Description

This command returns the bitwise XOR comparison of *E1* and *E2*.

a XOR b returns 1 if *a* is true or if *b* is true but not if both of them are the same value.

Example 139: XOR Operator

```
1 XOR 2  // Returns 3
2 XOR 2  // Returns 0
```

Check also

Operators Priority, p. 17; *AND, p. 24*; *OR, p. 85*

4 Debugging

The BASIC IDE comes with an integrated console.

This means that the debugging can be performed directly within the code.

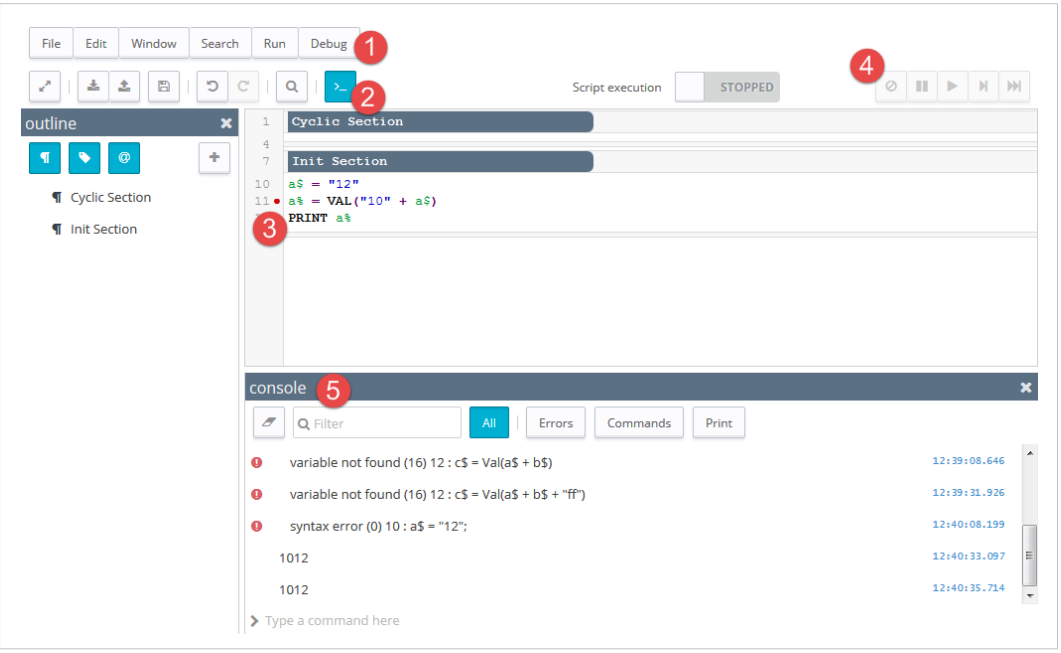

Fig. 3 Interface of the BASIC IDE

Debug interface of the BASIC IDE	
#	**Explanation**
1	This is the general menu for the debugging. It allows to • Pause, Continue and Abort • Perform step by step action • Remove all breakpoints
2	This icon shows / hides the console frame (number 5).
3	Manually point out the line the debugger should stop on.
4	Control the flow of the BASIC script: play/resume, pause, perform step by step action.
5	The console frame provides more advanced actions such as: • sort the different types of log (error, command or print) • see the result of functions, commands, … • manually trigger actions such as functions, label, ... Multi-lines debugging is not allowed.

5 BASIC Error Codes

The following table lists the error codes returned in the *ONERROR* command.

Error Code	Error Name
0	syntax error
1	“(or)” expected
2	no expression present
3	“=” expected
4	not a variable
5	invalid parameter
6	duplicate label
7	undefined label
8	THEN expected
9	TO expected
10	too many nested FOR loops
11	NEXT without FOR
12	too many nested GOSUBs
13	RETURN without GOSUB
14	out of memory
15	invalid var name
16	variable not found
17	unknown operator
18	mixed string&num operation
19	Dim index error
20	“,” expected
21	number expected
22	invalid assignment
23	quote too long
24	var or keyword too long
25	no more data
26	reenter timer
27	label not found
28	operation failed
29	ENDIF expected
30	ENDIF without IF
31	ELSE without IF
32	math error
33	IO Error
34	end of file
35	val in val

6 Configuration Fields

This section describes the fields that can be used in combination with the *GETSYS* and *SETSYS* commands. All the fields are readable and writable (unless specified otherwise).

Fields are divided as follow:

- System, User and Tag list can be found at the in theconfig.txt file in the

  Related Documents

- Communication can be found in the comcfg.txt file in the

  Related Documents

Before using parameters from one of the sections, it must first be loaded with the *SETSYS SYS, xxx*command

Example 140: Declare SETSYS before GETSYS

```
// Setting the identification of the device
// Printing the information
// Parameter = Identification, Information
SETSYS SYS, "LOAD"
SETSYS SYS, "Identification", "10.0.0.53"
PRINT GETSYS SYS, "Information"
SETSYS SYS, "SAVE"
```

© 2021 HMS Industrial Networks
Box 4126
300 04 Halmstad, Sweden

info@hms.se